Property-based testing for Spark Streaming (Extended version)*

Adrián Riesco and Juan Rodríguez-Hortalá

Technical Report SIC 02/18

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

May 2018

*This research has been partially supported by the Spanish MINECO projects CAVI-ART (TIN2013-44742-C4-3-R) and *TRACES* (TIN2015-67522-C3-3-R) and by the Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).

Abstract

Stream processing has reached the mainstream in the last years, as a new generation of open source distributed stream processing systems, designed for scaling horizontally on commodity hardware, has brought the capability for processing high volume and high velocity data streams to companies of all sizes. In this work we propose a combination of temporal logic and property-based testing (PBT) for dealing with the challenges of testing programs that employ this programming model. In particular we focus on testing Spark Streaming programs written with the Spark API for the functional language Scala, using the PBT library ScalaCheck. For that we add temporal logic operators to a set of new ScalaCheck generators and properties, as part of our testing library sscheck. We formalize our approach in a discrete time temporal logic for finite words, with some additions to improve the expressiveness of properties, which includes timeouts for temporal operators and a binding operator for letters.

Keywords: Property-based testing, Spark Streaming, Linear temporal logic, First-order modal logic, Scala

Contents

```
scala> val cs: RDD[Char] = sc.parallelize("let's count some letters", numSlices=3)
scala> cs.map{(_, 1)}.reduceByKey{_+_}.collect()
res4: Array[(Char, Int)] = Array((t,4), ( ,3), (l,2), (e,4), (u,1), (m,1), (n,1),
                                 (r,1), ('',1), (s,3), (o,2), (c,1))
```

Figure 1: Letter count in Spark

1 Introduction

With the rise of Big Data technologies [Marz and Warren, 2015], distributed stream processing systems (SPS) [Akidau et al., 2013, Marz and Warren, 2015] have gained popularity in the last years. This later generation of SPS systems, characterized by a distributed architecture designed for horizontal scaling, was pioneered by Internet-related companies, that had to find innovative solutions to scale their systems to cope with the fast growth of the Internet. These systems are used to continuously process high volume streams of data. One of the earliest examples is MillWheel [Akidau et al., 2013], which was designed and used by Google for tasks like anomaly detection and cluster-health monitoring. Twitter is also known for using SPS systems like Apache Storm [Marz and Warren, 2015] and its successor Twitter Heron [Ramasamy, 2015] which are employed to process the massive continuous flow of tweets in the Twitter Firehose, computing approximate statistics about tweets with latencies of seconds, in order to build data products like Twitter's trending topics. Yahoo's S4 [Neumeyer et al., 2010] was used for live parameter tuning of its search advertising system using the user traffic. LinkedIn built the data processing pipeline for its social network on top of Samza [Gorawski et al., 2014] and Kafka [Kreps, 2014]. However, the first precedents of stream processing systems come back as far as the early synchronous data-flow programming languages like Lutin [Raymond et al., 2008] or Lustre [Halbwachs, 1992].

A plethora of new distributed SPS have arisen in the last years, with proposals like Apache Flink [Schelter et al., 2013, Carbone et al., 2015a], Akka Streams [Kuhn and Allen, 2014], and Spark Streaming [Zaharia et al., 2013]. Among them Spark Streaming stands out as a particularly attractive option, due to the increasing maturity of the Spark ecosystem and its growing adoption in the industry. In this work we focus on Spark Streaming, and in particular on its Scala API. Spark [Zaharia et al., 2012] is a distributed processing engine that was designed as an alternative to Hadoop MapReduce [Marz and Warren, 2015], but with a focus on iterative processing—e.g. to implement distributed machine learning algorithms—and interactive low latency jobs—e.g. for ad hoc SQL queries on massive datasets. The key to achieving these goals is an extended memory hierarchy that allows for an increased performance in many situations, and a data model based on immutable collections inspired in functional programming that is the basis for its fault tolerance mechanism. The core of Spark is a batch computing framework [Zaharia et al., 2012] that is based on manipulating so called Resilient Distributed Datasets (RDDs), which provide a fault tolerant implementation of distributed collections. Computations are defined as transformations on RDDs, that should be deterministic and side-effect free, as the fault tolerance mechanism of Spark is based on its ability to recompute any fragment (partition) of an RDD when needed. Hence Spark programmers are encouraged to define RDD transformations that are pure functions from RDD to RDD, and the set of predefined RDD transformations includes typical higher-order functions like map, filter, etc., as well as aggregations by key and joins for RDDs of key-value pairs. We can also use Spark actions, which allow us to collect results into the *driver program* or store them into an external data store. The driver program is the local process that starts the connection to the Spark cluster, and issues the execution of Spark jobs, acting as a client of the Spark cluster. Spark actions are impure, so idempotent actions are recommended in order to ensure a deterministic behavior even in the presence of recomputations triggered by the fault tolerance or speculative task execution mechanisms [Apache Spark Team, 2016]. Spark is written in Scala and offers APIs for Scala, Java, Python, and R; in this work we focus on the Scala API. The example in Figure 1 uses the Scala Spark shell to implement a variant of the famous word count example that in this case computes the number of occurrences of each character in a sentence. For that we use `parallelize`, a feature of Spark that allows us to create an RDD from a local collection, which is useful for testing. We start with a set of chars distributed among 3 partitions, we pair each char with a 1 by using `map`, and then group by first component in the pair and sum by the second by using `reduceByKey` and the addition function (`_+_`), thus obtaining a set of (char, frequency) pairs. We collect this set into an `Array` in the driver with `collect`.

Besides the core RDD API, the Spark release contains a set of high level libraries that accelerates the development of Big Data processing applications, and that are also one of the reasons for its growing popularity. This includes libraries for scalable machine learning, graph processing, a SQL engine, and

```
object HelloSparkStreaming extends App {
  val conf = new SparkConf().setAppName("HelloSparkStreaming")
                            .setMaster("local[5]")
  val sc = new SparkContext(conf)
  val batchInterval = Duration(100)
  val ssc = new StreamingContext(sc, batchInterval)
  val batches = "let's count some letters, again and again"
                .grouped(4)
  val queue = new Queue[RDD[Char]]
  queue ++= batches.map(sc.parallelize(_, numSlices = 3))
  val css : DStream[Char] = ssc.queueStream(queue,
                                            oneAtATime = true)
  css.map{(_, 1)}.reduceByKey{_+_}.print()
  ssc.start()
  ssc.awaitTerminationOrTimeout(5000)
  ssc.stop(stopSparkContext = true)
}
```

```
-----------------------
Time: 1449638784400 ms
-----------------------
(e,1)
(t,1)
(l,1)
(',1)
...
-----------------------
Time: 1449638785300 ms
-----------------------
(i,1)
(a,2)
(g,1)
-----------------------
Time: 1449638785400 ms
-----------------------
(n,1)
```

Figure 2: Letter count in Spark Streaming

Spark Streaming, which is the focus of this work. In Spark Streaming, the notions of transformations and actions are extended from RDDs to DStreams (Discretized Streams), which are series of RDDs corresponding to splitting an input data stream into fixed time windows, also called micro batches. Micro batches are generated at a fixed rate according to the configured *batch interval*. Spark Streaming is synchronous in the sense that given a collection of input and transformed DStreams, all the batches for each DStream are generated at the same time as the batch interval is met. Actions on DStreams are also periodic and are executed synchronously for each micro batch. The code in Figure 2 is the streaming version of the code in Figure 1. In this case we process a DStream of characters, where batches are obtained by splitting a String into pieces by making groups (RDDs) of 4 consecutive characters. We use the testing utility class `QueueInputDStream`, which generates batches by picking RDDs from a queue, to generate the input DStream by parallelizing each substring into an RDD with 3 partitions. The program is executed using the local master mode of Spark, which replaces slave nodes in a distributed cluster by local threads, which is useful for developing and testing.

1.1 The problem of testing

Testing an SPS-based program is intrinsically hard, because it requires handling time and events. There are several proposals in the literature that tackle the problem of testing and modeling systems that deal with time. In this work we follow Pnueli's approach [Pnueli, 1986] that pioneered the usage of temporal logic for testing reactive systems. Our final goal is facilitating the adoption of temporal logic as an every day tool for testing SPS-based programs. But, how could we present temporal logic in a way accessible to the average software developer? Maybe using a software development technique that already exposes developers to first order logic, and thas has some adoption in the industry, would be a good idea. In this work we propose exploring how property-based testing (PBT) [Claessen and Hughes, 2011], as realized in ScalaCheck [Nilsson, 2014], can be used as a bridge between formal logic and software development practices like test-driven development (TDD) [Beck, 2003]. Classical unit testing with xUnit-like frameworks [Meszaros, 2007] is based on specifying input – expected output pairs, and then comparing the expected output with the actual output obtained by applying the test subject to the input. On the other hand, in PBT a test is expressed as a property, which is a formula in a restricted version of first order logic that relates program input and output. The testing framework checks the property by evaluating it against a bunch of randomly generated inputs. If a counterexample for the property is found then the test fails, otherwise it passes. This allows developers to obtain quite a good test coverage of the production code with a fairly small investment on development time, specially when compared to xUnit frameworks. However xUnit frameworks are still useful for testing corner cases that would be difficult to cover with a PBT property. The following is a "hello world" ScalaCheck property that checks the commutativity of addition:[1]

```
class HelloPBT extends Specification with ScalaCheck {
  def is = s2"""Hello world PBT spec,
                where int addition is commutative $intAdditionCommutative"""

  def intAdditionCommutative =
    Prop.forAll("x" |: arbitrary[Int], "y" |: arbitrary[Int]) { (x, y) =>
```

[1]Here we use the integration of ScalaCheck with the Specs2 [Torreborre, 2014] testing library.

```
      x + y === y + x
  }.set(minTestsOk = 100)
}
```

PBT is based on *generators* (the functions in charge of computing the inputs, which define the domain of discourse for a formula) and *assertions* (the atoms of a formula), which together with a *quantifier* form a *property* (the formula to be checked). In the example above the universal quantifier `Prop.forAll` is used to define a property that checks whether the assertion `x + y === y + x` holds for 100 values for `x` and `y` randomly generated by two instances of the integer generator `arbitrary[Int]`. Each of those pairs of values generated for `x` and `y` is called a *test case*, and a test case that refutes the assertions of a property is called a *counterexample*. Here `arbitrary` is a higher order generator that is able to generate random values for predefined and custom types. Besides universal quantifiers, ScalaCheck supports existential quantifiers—although these are not much used in practice [Nilsson, 2014, Venners, 2015]—, and logical operators to compose properties. PBT is a sound procedure to check the validity of the formulas implied by the properties, in the sense that any counterexample that is found can be used to build a definitive proof that the property is false. However, it is not complete, as there is no guarantee that the whole space of test cases is explored exhaustively, so if no counterexample is found then we cannot conclude that the property holds for all possible test cases that could had been generated: *all failing properties are definitively false, but not all passing properties are definitively true*. PBT is a lightweight approach that does not attempt to perform sophisticated automatic deductions, but it provides a very fast test execution that is suitable for the TDD cycle, and empirical studies [Claessen and Hughes, 2011, Shamshiri et al., 2015] have shown that in practice random PBT obtains good results, with a quality comparable to more sophisticated techniques. This goes in the line of assuming that in general testing of non trivial systems is often incomplete, as the effort of completely modeling all the possible behaviors of the system under test with test cases is not cost effective in most software development projects, except for critical systems.

1.1.1 Property-based testing with temporal operators.

Thanks to PBT, we currently have developers using first order logic to specify test cases, so to realize our proposal all that is left is adding temporal operators to the mix. We have implemented this idea in a library that extends ScalaCheck with temporal logic operators. Our logic includes classical linear temporal logic (LTL) [Blackburn et al., 2006] operators with bounded time such as *always* φ *in* t, which indicates that φ must hold for the next t instants, or φ *until* ψ *in* t, which indicates that φ currently holds and, before t instants of time elapse, ψ must hold. This way we obtain a propositional LTL formula extended with an outer universal quantifier over the test cases produced by the generators. Our temporal uses *discrete time*, as DStreams are discrete. Also, the logic should fit the simple property checking mechanism of PBT, that requires fast evaluation of test cases. For this reason we use a temporal logic for finite words, like those used in the field of runtime verification [Leucker and Schallhart, 2009], instead of using infinite ω-words as usual in model checking. Although any Spark DStream is supposed to run indefinitely, so it might well be modeled by an infinite word, in our setting we only model a finite prefix of the DStream. This follows the idea of PBT and allows us to implement a simple, fast, and sound procedure for evaluating test cases, because if a prefix of a DStream refutes a property then the whole infinite DStream refutes the property as well. On the other hand the procedure is not complete because only a prefix of the DStream is evaluated, but anyway PBT was not complete in the first place. Hence a *test case* will be a tuple of *finite prefixes* of DStreams, that corresponds to a *finite word* in this logic, and the aforementioned external quantifier ranges over the domain of finite words. In Section 2.1 there is a precise formulation of our logic LTL_{ss}, for now let's consider a concrete example in order to get a quick grasp of our proposal.

Example 1 *We would like to test a Spark Streaming program that receives a stream of events describing user activity and returns a stream with the identifiers of banned users. To keep the example simple, we assume that the input records are pairs containing a* `Long` *user id, and a* `Boolean` *value indicating whether the user has been honest at that instant. The output stream should include the ids of all those users that have been malicious now or in a previous instant. So, the test subject that implements it has type* `testSubject: DStream[(Long, Boolean)] => DStream[Long])`*. Note that a trivial, stateless implementation of this behavior that just keeps the first element of the pair should fail to achieve this goal, as it is not able to remember which users had been malicious in the past:*

```
def statelessListBannedUsers(ds: DStream[(Long, Boolean)]): DStream[Long] =
  ds.map(_._1)
```

To define a property that captures the expected behavior, we start by defining a generator *for (finite prefixes of) the input stream. As we want this input to change with time, we use a temporal logic formula to specify the generator. We start by defining the atomic non-temporal propositions, which are generators of micro batches with type* `Gen[Batch[(Long, Boolean)]]`*, where* `Batch` *is a class extending* `Seq` *that represents a micro batch. We can generate good batches, where all the users are honest, and bad batches, where a user has been malicious. We generate batches of* `20` *elements, and use* `15L` *as the id for the malicious id:*

```
val batchSize = 20
val (badId, ids) = (15L, Gen.choose(1L, 50L))
val goodBatch = BatchGen.ofN(batchSize, ids.map((_, true)))
val badBatch = goodBatch + BatchGen.ofN(1, (badId, false))
```

So far generators are oblivious to the passage of time. But in order to exercise the test subject thoroughly, we want to ensure that a bad batch is indeed generated, and that several arbitrary batches are generated after it, so we can check that once a user is detected as malicious, it is also considered malicious in subsequent instants. Moreover, we want all this to happen within the confines of the generated finite DStream prefix. This is where timeouts *come into play. In our temporal logic we associate a timeout to each temporal operator, that constrains the time it takes for the operator to resolve. For example in a use of* until *with a timeout of t, the second formula must hold before t instants have passed. Translated to generators this means that in each generated DStream prefix a batch for the second generator is generated before t batches have passed, i.e. between the first and the t-th batch. This way we facilitate that the interesting events had enough time to happen during the limited fraction of time considered during the evaluation of the property:*

```
val (headTimeout, tailTimeout, nestedTimeout) = (10, 10, 5)
val gen = BatchGen.until(goodBatch, badBatch, headTimeout) ++
          BatchGen.always(Gen.oneOf(goodBatch, badBatch), tailTimeout)
```

where `BatchGen` *is the class for representing batches of elements a discrete data stream. The resulting generator* `gen` *has type* `Gen[PDStream[(Long, Boolean)]]`*, where* `PDStream` *is a class that represents sequences of micro batches corresponding to a DStream prefix. Here* `headTimeout` *limits the number of batches before the bad batch occurs, while* `tailTimeout` *limits the number of arbitrary batches generated after that. The output stream is simply the result of applying the test subject to the input stream. Now we define the* assertion *that completes the property, as a* temporal logic formula*:*

```
type U = (RDD[(Long, Boolean)], RDD[Long])
val (inBatch, outBatch) = ((_: U)._1, (_: U)._2)
val formula = {
  val allGoodInputs = at(inBatch)(_ should foreachRecord(_._2 == true))
  val badInput = at(inBatch)(_ should existsRecord(_ == (badId, false)))
  val noIdBanned = at(outBatch)(_.isEmpty)
  val badIdBanned = at(outBatch)(_ should existsRecord(_ == badId))

  ((allGoodInputs and noIdBanned) until badIdBanned on headTimeout) and
  (always { badInput ==> (always(badIdBanned) during nestedTimeout) }
           during tailTimeout)  }
```

Atomic non-temporal propositions correspond to assertions on the micro batches for the input and output DStreams. We use an idiom where the function `at` *below is used with a projection function like* `inBatch` *or* `outBatch` *to apply an assertion on part of the current letter, e.g. the batch for the current input. The assertions* `foreachRecord` *and* `existsRecord` *are custom Specs2 assertions that allow to check whether a predicate holds or not for all or for any of the records in an RDD, respectively. This way we are able to define non-temporal atomic propositions like* `allGoodInputs`*, that states that all the records in the input DStream correspond to honest users.*

But we know that `allGoodInputs` *will not be happening forever, because* `gen` *eventually creates a bad batch, so we combine the atomic propositions using temporal operators to state things like "we have good inputs and no id banned until we ban the bad id" and "each time we get a bad input we ban the bad id for some time." Here we use the same timeouts we used for the generators, to enforce the formula within the time interval where the interesting events are generated. Also, we use an additional* `nestedTimeout` *for the nested* `always`*. Timeouts for operators that apply an universal quantification on time, like* always*, limit*

the number of instants that the quantified formula needs to be true for the whole formula to hold. In this case we only have to check `badIdBanned` *for* `nestedTimeout` *batches for the nested* `always` *to be evaluated to true. Following ideas from the field of runtime verification [Bauer et al., 2006, Bauer et al., 2007], we consider a 3-valued logic where the third value corresponds to an inconclusive result used as the last resort when the input finite word is consumed before completely solving the temporal formula. Timeouts for universal time quantifiers help relaxing the formula so its evaluation is conclusive more often, while timeouts for existential time quantifiers like* until *make the formula more strict. We consider that it is important to facilitate expressing properties with a definite result, as quantifiers like* exists*, that often lead properties to an inconclusive evaluation, have been abandoned in practice by the PBT user community [Nilsson, 2014, Venners, 2015].*

Finally, we use our temporal universal quantifier `forAllDStream` *to put together the temporal generator and formula, getting a property that checks the formula for all the finite DStreams prefixes produced by the generator:*

```
forAllDStream(gen)(testSubject)(formula).set(minTestsOk = 20)
```

The property fails as expected for the faulty trivial implementation above, and succeeds for a correct stateful implementation (see Appendix **??** *for details).*

We carried out these ideas on the testing library sscheck [Riesco and Rodríguez-Hortalá, 2017], that we previously presented in a tool paper [Riesco and Rodríguez-Hortalá, 2016b], and in a leading engineering conference [Riesco and Rodríguez-Hortalá, 2016a]. The present paper extends that work by:

- Improving the logic by (i) redefining the semantics of formulas using a first order structure for letters, that are evaluated under a given interpretation, (ii) introducing a new operator that allows us to bind the content and the time in the current batch, (iii) redefining the previous results for the new logic, and (iv) defining a new recursive definition that allows us to simplify formulas in a lazy way.
- Formally proving the theoretical results arising from the new formulation.
- Formalizing the generation of words from formulas.
- Enhancing the performance of the tool by using the laziness features of Scala to take advantage of the step-wise transformation defined for the logic.
- Providing extensive examples of sscheck properties, including safety and liveness properties.

The rest of the paper is organized as follows: Section 2 describes our logic for testing stream processing systems, while Section 3 presents its implementation for Spark. Section 4 discusses some related work. Finally, Section 5 concludes and presents some subjects of future work.

2 A Logic for Testing Stream Systems

We present in this section our linear temporal logic for defining properties on Spark Streaming programs. We first define the basics of the logic, then we show an stepwise formula evaluation procedure that is the basis for our prototype, and finally we formalize the generation of test cases from formulas.

2.1 A Linear Temporal Logic with Timeouts for practical specification of stream processing systems

The basis of our proposal in the LTL_{ss} logic, a linear temporal logic that combines and specializes both LTL_3 [Bauer et al., 2006] and First-order Modal Logic [Fitting and Mendelsohn, 1998], borrowing some ideas from TraceContract [Barringer and Havelund, 2011]. LTL_3 is an extension of LTL [Alur and Henzinger, 1994] for runtime verification that takes into account that only *finite* executions can be checked, and hence a new value ? (inconclusive) can be returned in case a property cannot be effectively evaluated to either *true* ($\top$) or *false* ($\bot$) in the given execution, because the word considered was too short. These values form a lattice with $\bot \leq ? \leq \top$; we remind how the logical connectives work in this case in Table 1. LTL_{ss} uses the same domain as LTL_3 for evaluating formulas, and the same truth tables for the basic non-temporal logical connectives. LTL_{ss} is also influenced by First-order Modal Logic, an extension to First-order of the standard propositional modal logic approach [Blackburn et al., 2006]. Although the

	$\vee$	$\bot$	?	$\top$	$\wedge$	$\bot$	?	$\top$	$\neg$
$\bot$		$\bot$	?	$\top$		$\bot$	$\bot$	$\bot$	$\top$
?		?	?	$\top$		$\bot$	?	?	?
$\top$		$\top$	$\top$	$\top$		$\bot$	?	$\top$	$\bot$

Table 1: Truth tables for the logical connectives in LTL_3

propositional approach in [Riesco and Rodríguez-Hortalá, 2016b] was enough for generating new values and dealing with some interesting properties —including safety properties— we noticed that some other properties involving variables bound in previous letters —e.g. some liveness properties— could not be easily specified in our logic. For this reason we have extended the original version of LTL_{ss} with a binding operator inspired by a similar construction from TraceContract [Barringer and Havelund, 2011], which provides a form of universal quantification over letters, that makes it easy to define liveness properties, as we will see in Section 3.2.4.

Formulae Syntax We assume a denumerable set $\mathcal{V}$ of variables $(x, y, z, o, \ldots)$, a denumerable set $\mathcal{P}$ of predicate symbols $(p, q, r, \ldots)$ with associated arity—with $\mathcal{P}^n$ the set of predicate symbols with arity n, and $\mathbb{N} \subseteq \mathcal{F}^0$—, and a denumerable set $\mathcal{F}$ of function symbols $(f, g, h, \ldots)$ with associated arity—with $\mathcal{F}^n$ the set of function symbols with arity n. Then, terms $e \in \mathit{Term}$ are built as:

$$\mathit{Term} \ni e ::= \quad x \mid f(e_1, \ldots, e_n) \text{ for } x \in \mathcal{V}, f \in \mathcal{F}^n, e_1, \ldots, e_n \in \mathit{Term}$$

Typically, propositional formulations of LTL [Alur and Henzinger, 1994] consider words that use the power set of atomic propositions as its alphabet. However, we consider the alphabet $\Sigma = \mathit{Term} \times \mathbb{N}$ of timed terms. Over this alphabet we define the set of finite words Σ^*, i.e. finite sequences of timed terms. We use ϵ for the empty word, and the notation $u = u_1 \ldots u_n$ to denote that $u \in \Sigma^*$ has length $\mathit{len}(u)$ equal to n, and u_i is the letter at position i in u. Each letter $u_i \equiv (e_i, t_i)$ corresponds to the term e_i that can be observed at instant i after t_i units of time have been elapsed. For example, for a Spark Streaming program with one input DStream and one output DStream, the term e_i would correspond to a pair of RDDs, one representing the micro batch for the input DStream at time t_i, and another the micro batch for the output DStream at time t_i.
It is important to distinguish between the instant i, which refers to logic time and can be understood as a "counter of states," and t_i, which refers to real time. This real time satisfies the usual condition of *monotonicity* $(t_i \leq t_{i+1}, i \geq 0)$, but does not satisfy *progress* $(\forall t \in \mathbb{N}, \exists_{i \geq 0} t_i > t)$, since we work with finite words. It is also important to note that time is discrete but the time between successive states may be arbitrary. Also note the condition $\mathbb{N} \subseteq \mathcal{F}^0$ above ensures that the interpretations used to evaluate formulas later in this section, are always able to provide a meaning for the time literals. The set of LTL_{ss} formulas LTL_{ss} is defined as follows:

$$\begin{aligned} LTL_{ss} \ni \varphi ::= \quad & \bot \mid \top \mid p(e_1, \ldots, e_n) \mid e_1 = e_2 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \\ & X\varphi \mid \Diamond_t \varphi \mid \Box_t \varphi \mid \varphi \; U_t \; \varphi \mid \varphi \; R_t \; \varphi \mid \lambda^o_x.\varphi \end{aligned}$$

We will use the notation $X^n\varphi, n \in \mathbb{N}^+$, as a shortcut for n applications of the operator X to φ. Although we provide a precise formulation for the interpretation of these formulas later in this section, the underlying intuitions are as follows:

- The first eight formulas are based on classical first order non-temporal logical connectives, including contradiction, tautology, atomic formulas based on predicate application and equality, and the negation and the usual binary connectives.
- $X\varphi$, read "next φ", indicates that the formula φ should hold in the next state.
- $\Diamond_t \varphi$, read "eventually φ in t," indicates that φ holds in any of the next t states (including the current one).
- $\Box_t \varphi$, read "always φ in t," indicates that φ holds in all of the next t states (including the current one).
- $\varphi_1 \; U_t \; \varphi_2$, read "$\varphi_1$ holds until φ_2 in t," indicates that φ_1 holds until φ_2 holds in the next t states, including the current one, and φ_2 must hold eventually. Note that it is enough for φ_1 to hold until the state previous to the one where φ_2 holds.

- $\varphi_1\ R_t\ \varphi_2$, read "$\varphi_2$ is released by φ_1 in t," indicates that φ_2 holds until both φ_1 and φ_2 hold in the next t states, including the current one. However, if φ_1 never holds and φ_2 always holds the formula holds as well.

- $\lambda^o_x.\varphi$, read "consume the current letter to produce φ", indicates that given (e, t) the letter for the current state, then the formula resulting from replacing in φ the variables x and o by e and t, respectively, should hold in the next state. We call this the *consume* operator.

We say that a formula is *timeless* when it does not contain any of the temporal logical connectives. An LTL_{ss} formula or term is closed or ground if it has no free variables. Free variables, that can also appear in temporal connectives, are computed by the function fv as follows:

$$
\begin{array}{lcl}
fv(\bot) & = & \emptyset \\
fv(\top) & = & \emptyset \\
fv(x) & = & \{x\} \\
fv(f(e_1, \ldots, e_n)) & = & fv(e_1)\ \cup\ \ldots\ \cup\ fv(e_n) \\
fv(p(e_1, \ldots, e_n)) & = & fv(e_1)\ \cup\ \ldots\ \cup\ fv(e_n) \\
fv(t_1 = t_2) & = & fv(t_1)\ \cup\ fv(t_2) \\
fv(\neg\varphi) & = & fv(\varphi) \\
fv(\varphi_1 \vee \varphi_2) & = & fv(\varphi_1)\ \cup\ fv(\varphi_2) \\
fv(\varphi_1 \wedge \varphi_2) & = & fv(\varphi_1)\ \cup\ fv(\varphi_2) \\
fv(\varphi_1 \rightarrow \varphi_2) & = & fv(\varphi_1)\ \cup\ fv(\varphi_2) \\
fv(X\varphi) & = & fv(\varphi) \\
fv(\Diamond_t\varphi) & = & fv(t)\ \cup\ fv(\varphi) \\
fv(\Box_t\varphi) & = & fv(t)\ \cup\ fv(\varphi) \\
fv(\varphi_1\ U_t\ \varphi_2) & = & fv(\varphi_1)\ \cup\ fv(\varphi_2)\ \cup\ fv(t) \\
fv(\varphi_1\ R_t\ \varphi_2) & = & fv(\varphi_1)\ \cup\ fv(\varphi_2)\ \cup\ fv(t) \\
fv(\lambda^o_x.\varphi) & = & fv(\varphi)\backslash\{x, o\}
\end{array}
$$

We will only consider closed formulas in the following. Moreover, we will use the notation $\varphi[b \mapsto v_1, r \mapsto v_2] \equiv (\varphi[b \mapsto v_1])[r \mapsto v_2]$ to indicate that b and r are substituted by v_1 and v_2, respectively. Assuming that all variables in universal quantifiers are different under the usual alpha-conversion assumption, substitutions are defined as follows:[2]

$$
\begin{array}{lcl}
\bot[x \mapsto v] & = & \bot \\
\top[x \mapsto v] & = & \top \\
f(e_1, \ldots, e_n)[x \mapsto v] & = & f(e_1[x \mapsto v], \ldots, e_n[x \mapsto v]) \\
p(e_1, \ldots, e_n)[x \mapsto v] & = & p(e_1[x \mapsto v], \ldots, e_n[x \mapsto v]) \\
(e_1 = e_2)[x \mapsto v] & = & e_1[x \mapsto v] = e_2[x \mapsto v] \\
(\neg\varphi)[x \mapsto v] & = & \neg(\varphi[x \mapsto v]) \\
(\varphi_1 \vee \varphi_2)[x \mapsto v] & = & \varphi_1[x \mapsto v]\ \vee \varphi_2[x \mapsto v] \\
(\varphi_1 \wedge \varphi_2)[x \mapsto v] & = & \varphi_1[x \mapsto v]\ \wedge \varphi_2[x \mapsto v] \\
(\varphi_1 \rightarrow \varphi_2)[x \mapsto v] & = & \varphi_1[x \mapsto v]\ \rightarrow \varphi_2[x \mapsto v] \\
(X\varphi)[x \mapsto v] & = & X(\varphi[x \mapsto v]) \\
(\Diamond_t\varphi)[x \mapsto v] & = & (\Diamond_{t[x \mapsto v]}\varphi[x \mapsto v]) \\
(\Box_t\varphi)[x \mapsto v] & = & (\Box_{t[x \mapsto v]}\varphi[x \mapsto v]) \\
(\varphi_1\ U_t\ \varphi_2)[x \mapsto v] & = & (\varphi_1[x \mapsto v]\ U_{t[x \mapsto v]}\ \varphi_2[x \mapsto v]) \\
(\varphi_1\ R_t\ \varphi_2)[x \mapsto v] & = & (\varphi_1[x \mapsto v]\ R_{t[x \mapsto v]}\ \varphi_2[x \mapsto v]) \\
(\lambda^o_y.\varphi)[x \mapsto v] & = & \lambda^o_y.(\varphi[x \mapsto v]) \\
x[x \mapsto v] & = & v \\
x_1[x_2 \mapsto v] & = & x_1 \text{ if } x_1 \neq x_2
\end{array}
$$

Logic for finite words In order to evaluate our formulas, we need a way to interpret the timed terms that we use as the alphabet. In line with classical formulations of first order Boolean logic [Smullyan, 1995], formulas are evaluated in the context of an *interpretation structure* $\mathcal{A}$, which is a pair (A, I) where A is a non-empty set that is used as the interpreting domain, and I is an interpretation function that assigns to each $f \in \mathcal{F}^n$ an interpreting function $I(f) : A^n \rightarrow A$, and to each $p \in \mathcal{P}^n$ an interpreting relation $I(p) \subseteq A^n$. These interpretations are naturally applied to closed terms by induction

[2]Note that the substitution is applied to the timeout in quantifiers. Since general timeouts are arithmetic expressions substitutions are applied as usual; we skip these equations for the sake of simplicity.

on the structure of terms as $[\![f(e_1,\ldots,e_n)]\!]^{\mathcal{A}} = I(f)([\![e_1]\!]^{\mathcal{A}},\ldots,[\![e_1]\!]^{\mathcal{A}}) \in A$. Our logic proves *judgments* of the form $u, i \vDash^{\mathcal{A}} \varphi : v$ that state that considering the finite word $u \in \Sigma^*$ from the position of its i-th letter, the formula $\varphi \in LTL_{ss}$ is evaluated to the truth value $v \in \{\top, \bot, ?\}$—where ? stands for inconclusive—under the interpretation $\mathcal{A}$. In other words, if we stand at the i-th letter of u and start evaluating φ, moving forward in u one letter at a time as time progresses, and using $\mathcal{A}$ to interpret the terms that appear in the word and in the formula, we end up getting the truth value v. Note that in our judgments the same interpretation structure holds "eternally" constant for all instants, while only one letter of u is occurring at each instant. This is modeling what happens during the testing of a Spark Streaming job, and in general of any program: the code that defines how the program reacts to its inputs is the same during the execution of the program—which is modeled by a constant interpretation structure—, while the inputs of the program and their corresponding output change with time —which is modeled by the sequence of letters that is the word. That is not able to model updates in the program code, but is expressive enough to be used during unit and integration testing, where the program code is fixed. Note the predicate symbols used in the formula correspond to the assertions used in the tests [Torreborre, 2014], whose meaning is also constant during the test execution. Judgments are defined by the following rules, where only the first rule that fits should be applied, and we assume $\mathcal{A} = (A, I)$:[3]

$$u, i \vDash^{\mathcal{A}} v : v \quad \text{if } v \in \{\bot, \top\}$$

$$u, i \vDash^{\mathcal{A}} p(e_1,\ldots,e_n) : \begin{cases} \top & \text{if } ([\![e_1]\!]^{\mathcal{A}},\ldots,[\![e_n]\!]^{\mathcal{A}}) \subseteq I(p) \\ \bot & \text{otherwise} \end{cases}$$

$$u, i \vDash^{\mathcal{A}} e_1 = e_2 : \begin{cases} \top & \text{if } [\![e_1]\!]^{\mathcal{A}} = [\![e_2]\!]^{\mathcal{A}} \\ \bot & \text{otherwise} \end{cases}$$

$$u, i \vDash^{\mathcal{A}} \lambda^o_x.\varphi : \begin{cases} v & \text{if } i \leq len(u) \wedge u, i+1 \vDash^{\mathcal{A}} \varphi[x \mapsto e_i, o \mapsto t_i] : v \text{ for } u_i \equiv (e_i, t_i) \\ ? & \text{otherwise} \end{cases}$$

$$u, i \vDash^{\mathcal{A}} X\varphi : v \quad \text{if } u, i+1 \vDash^{\mathcal{A}} \varphi : v$$

$$u, i \vDash^{\mathcal{A}} \Diamond_t \varphi : \begin{cases} \top & \text{if } \exists k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi : \top \\ \bot & \text{if } \forall k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi : \bot \\ ? & \text{otherwise} \end{cases}$$

$$u, i \vDash^{\mathcal{A}} \Box_t \varphi : \begin{cases} \top & \text{if } \forall k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi : \top \\ \bot & \text{if } \exists k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi : \bot \\ ? & \text{otherwise} \end{cases}$$

$$u, i \vDash^{\mathcal{A}} \varphi_1\ U_t\ \varphi_2 : \begin{cases} \top & \text{if } \exists k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi_2 : \top \wedge \\ & \quad \forall j \in [i, k).\ u, j \vDash^{\mathcal{A}} \varphi_1 : \top \\ \bot & \text{if } \exists k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi_1 : \bot \wedge \\ & \quad \forall j \in [i, k].\ u, j \vDash^{\mathcal{A}} \varphi_2 : \bot \\ \bot & \text{if } \forall k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi_1 : \top \wedge \\ & \quad \forall l \in [i, min(i+(t-1), len(u))].\ u, l \vDash^{\mathcal{A}} \varphi_2 : \bot \\ ? & \text{otherwise} \end{cases}$$

$$u, i \vDash^{\mathcal{A}} \varphi_1\ R_t\ \varphi_2 : \begin{cases} \top & \text{if } \exists k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi_1 : \top \wedge \\ & \quad \forall j \in [i, k].\ u, j \vDash^{\mathcal{A}} \varphi_2 : \top \\ \top & \text{if } \forall k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi_2 : \top \\ \bot & \text{if } \exists k \in [i, i+(t-1)].\ u, k \vDash^{\mathcal{A}} \varphi_2 : \bot \wedge \\ & \quad \forall j \in [i, k).\ u, j \vDash^{\mathcal{A}} \varphi_1 : \bot \\ ? & \text{otherwise} \end{cases}$$

We say $u \vDash^{\mathcal{A}} \varphi$ iff $u, 1 \vDash^{\mathcal{A}} \varphi : \top$. The intuition underlying these definitions is that, if the word is too short to check all the steps indicated by a temporal operator and neither $\top$ or $\bot$ can be obtained before finishing the word, then ? is obtained. Otherwise, the formula is evaluated to either $\top$ or $\bot$ just by checking the appropriate sub-word. Note the consume operator (λ^o_x) is the only one that accesses the word directly, and that consume is equivalent to next applied to the corresponding formula at its

[3] Non-temporal operators follow the rules in Table 1.

body: for example $0\ \epsilon, 1 \vDash^{\mathcal{A}} \lambda^o_x.x = 0 : v \iff 0\ \epsilon, 1 \vDash^{\mathcal{A}} X(0 = 0) : v$. It is trivial to check that timeless formulas—i.e. without temporal connectives—are always evaluated to one of the usual binary truth values $\top$ of $\bot$, and that timeless formulas are evaluated to the same truth value irrespective of the word u and the position i considered, even for $u \equiv \epsilon$ or $i > len(u)$. As a consequence of this, some temporal formulas are true even for words with a length smaller than the number of letters referred by the temporal connectives in the formula: for example, for any i and $\mathcal{A}$ we have $\epsilon, i \vDash^{\mathcal{A}} X\top : \top$—*next* inspects the second letter, but the formula is true for the empty word because the body is trivially true—, $0\ 1\ \epsilon, i \vDash^{\mathcal{A}} \square_{10}\ (0 == 0) : \top$—this *always* refers to 10 letters, but it holds for a word with just 2 letters because the body is a tautology—, and similarly $0\ \epsilon, 1 \vDash^{\mathcal{A}} \lambda^o_x.(x = 0) : \top$ because $0\ \epsilon, 2 \vDash^{\mathcal{A}} 0 = 0 : \top$.

The resulting logic gives some structure to letters and words, but it is not fully a first order logic because it does not provide neither existential or universal quantifiers for words. The *consume* operator is somewhat a universal quantifier for letters, but can also be understood as a construct for parameter passing, like the binding operator from TraceContract [Barringer and Havelund, 2011] that this operator is modeled after.

Let us consider some example judgments for simple formulas, to start tasting the flavor of this logic.

Example 2 *Assume the set of constants $\{a, b, c\} \subseteq \mathcal{F}^0$, the set of variables $\{x, y, z, o, p, q\}$, and an interpretation structure (A, I) where $A = \mathcal{F}^0$ and $\forall f \in \mathcal{F}^0.\ I(f) = f$. Then for the word $u \equiv \boxed{(b,0)}\ \boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)}$ we can construct the following formulas:*

- $u \vDash^{\mathcal{A}} \Diamond_4\ \lambda^o_x.x = c : \bot$*, since c does not appear in the first four letters.*
- $u \vDash^{\mathcal{A}} \Diamond_5\ \lambda^o_x.x = c : ?$*, since we have consumed the word, c did not appear in those letters but the timeout has not expired.*
- $u \vDash^{\mathcal{A}} \square_4\ \lambda^o_x.(x = a \vee x = b) : \top$*, since either a or b is found in the first four letters.*
- $u \vDash^{\mathcal{A}} \square_5\ \lambda^o_x.(x = a \vee x = b) : ?$*, since the property holds until the word is consumed, but the user required more steps.*
- $u \vDash^{\mathcal{A}} \square_5\ \lambda^o_x.x = c : \bot$*, since the proposition does not hold in the first letter.*
- $u \vDash^{\mathcal{A}} \lambda^o_x.x = b\ U_2\ \lambda^p_y.y = a : \bot$*, since a appears in the third letter, but the user wanted to check just the first two letters.*
- $u \vDash^{\mathcal{A}} \lambda^o_x.x = b\ U_5\ \lambda^p_y.y = a : \top$*, since a appears in the third letter and, before that, b appeared in all the letters.*
- $u \vDash^{\mathcal{A}} \lambda^o_x.x = a\ R_2\ \lambda^p_y.y = b : \top$*, since b appears in all the required letters.*
- $u \vDash^{\mathcal{A}} \lambda^o_x.x = a\ R_4\ \lambda^p_y.y = b : \bot$*, since a appears in the third letter but b should appear as well.*
- $u \vDash^{\mathcal{A}} \square_3(\lambda^o_x.x = a) \to X(\lambda^p_y.y = a) : \top$*, since the formula holds in the first three letters (note that the fourth letter is required, since the formula involves the next operator).*
- $u \vDash^{\mathcal{A}} \square_4(\lambda^o_x.x = a) \to (\lambda^p_y.y = a) : ?$*, since we do not know what happens in the fifth letter, which is required to check the formula in the fourth instant.*
- $u \vDash^{\mathcal{A}} \square_2(\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a) : \bot$*, since in the first letter we have b but we do not have a until the third letter.*
- $u \vDash^{\mathcal{A}} (\lambda^o_x.x = b)\ U_2\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a) : \top$*, since $X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)$ holds in the second letter (that is, $(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)$ holds in the third letter, which can be also understood as a appears in the third and fourth letters).*
- $u \vDash^{\mathcal{A}} \lambda^o_x.\square_{o+6}\ x = b : \top$*, since the first letter is b and hence the equality is evaluated to $\top$.*

By using functions with arity greater than 0, and predicate symbols, we can construct more complex formulas. For example given $\mathbb{N} \subseteq \mathcal{F}^0, plus \in \mathcal{F}^2, leq \in \mathcal{P}^2$ and an interpretation structure (A, I) where $A = \mathbb{N}$, $\forall n \in \mathbb{N}.\ I(n) = n$, $I(plus)(x, y) = x + y$, $I(leq) = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x \leq y\}$, then we have

$$\boxed{(0,0)}\ \boxed{(1,2)}\ \boxed{(2,3)} \vDash^{\mathcal{A}} \Diamond_2\ \lambda^{o_1}_x.\lambda^{o_2}_y.leq(5, plus(x, y) : \top$$

For some examples in this paper we will assume *the Spark interpretation structure* $\mathcal{A}^{\mathcal{S}}$, that captures the observable semantics of a Spark program, and where timestamps are interpreted as Unix timestamps as usual in Java.[4] We will not provide a formalization of $\mathcal{A}^{\mathcal{S}}$, but the idea is that the prototype we present in Section 3 is intended to implement a procedure to prove judgments under the Spark interpretation structure. This interpretation assumes that letters are timed tuples of terms, and that each input or output DStream has an assigned tuple index, so that each element of the tuple represents the micro batch at that instant for the corresponding DStream. This is expressive enough to express any Spark Streaming program, because the set of DStreams is fixed during the lifetime of a Spark Streaming application.[5] Let us see some simple formulas we can write with this logic and how they are expressed in our prototype.

Example 3 *Assuming a Spark Streaming program with one input DStream and one output DStream, the following formula expresses the requirement that the output DStream will always contain numbers greater than 0, for 10 batches. Here we assume that* $nth2 \in \mathcal{F}^2$ *is interpreted in* $\mathcal{A}^{\mathcal{S}}$ *as a projection function that returns the second element of a tuple, and that* $allValuesGtZero \in \mathcal{P}^1$ *is a predicate that holds for those RDDs that only contain numbers that are greater than 0.*

$$\Box_{10}\ \lambda^{o}_{x}.allValuesGtZero(nth2(x))$$

This formula can be written in our prototype as follows:

```
always(nowTime[U]{ (letter, time) =>
  letter._2 should foreachRecord { _ > 0}
}) during 10
```

Assuming that $\mathcal{A}^{\mathcal{S}}$ *interprets the predicate* $leq \in \mathcal{P}^2$ *as the* $\leq$ *comparison operator, the following formula expresses that time always increases monotonically during 10 instants:*

$$\Box_{9}\ \lambda^{o_1}_{x_1}.\lambda^{o_2}_{x_2}.leq(o_1, o_2)$$

which we can express in our prototype as:

```
always(nextTime[U]{ (letter, time) =>
  nowTime[U]{ (nextLetter, nextTime) =>
    time.millis <= nextTime.millis
  }
}) during 9
```

Once the formal definition has been presented, we require a decision procedure for evaluating formulas. Next we present a formula evaluation algorithm inferred from the logic presented above.

Decision procedure for LTL_{ss} Just like ScalaCheck [Nilsson, 2014] and any other testing tool of the QuickCheck family [Claessen and Hughes, 2011, Papadakis and Sagonas, 2011], this decision procedure does not try to be complete for proving the veritative value of formulae, but just to be complete for failures, i.e. judgments to the truth value $\bot$. For this purpose we define an abstract rewriting system for reductions $u \vDash^{\mathcal{A}} \varphi \rightsquigarrow^{*} v$ for v in the same domain as above. We write $u \vDash \varphi \rightsquigarrow^{*} v$ when the interpretation $\mathcal{A}$ is implied by the context. Given a letter $a \in \Sigma$, a word $u \in \Sigma^{*}$, a set of terms $e, e_1, \ldots, e_n \in Term$, a timeout $t \in \mathbb{N}^{+}$, and formulas $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$, we have the following rules:[6]

1. Rules for $u \vDash^{\mathcal{A}} p(e_i)$:
   1) $u \vDash^{\mathcal{A}} p(e_1, \ldots, e_n) \rightsquigarrow \top$ if $([\![e_1]\!]^{\mathcal{A}}, \ldots, [\![e_n]\!]^{\mathcal{A}}) \subseteq I(p)$
   2) $u \vDash^{\mathcal{A}} p(e_1, \ldots, e_n) \rightsquigarrow \bot$ otherwise
2. Rules for $u \vDash e_1 = e_2$:
   1) $u \vDash^{\mathcal{A}} e_1 = e_2 \rightsquigarrow [\![e_1]\!]^{\mathcal{A}} = [\![e_2]\!]^{\mathcal{A}}$

[4] As interpreted both in Spark in particular (see `https://spark.apache.org/docs/1.6.2/api/scala/index.html#org.apache.spark.streaming.Time`) and in JVM languages in general (see `https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#currentTimeMillis--`).

[5] Because "once a context has been started, no new streaming computations can be set up or added to it," see `https://spark.apache.org/docs/1.6.2/streaming-programming-guide.html#initializing-streamingcontext`

[6] Formulas built with propositional operators just evaluate the sub-formulas and apply the connectives as shown in Table 1.

3. Rules for $u \vDash \lambda^o_x.\varphi$:
   1) $\epsilon \vDash \lambda^o_x.\varphi \quad \rightsquigarrow \quad ?$
   2) $(e,t)u \vDash \lambda^o_x.\varphi \quad \rightsquigarrow \quad u \vDash \varphi[x \mapsto e][o \mapsto t]$

4. Rules for $u \vDash X\ \varphi$:
   1) $\epsilon \vDash X\ \varphi \quad \rightsquigarrow \quad \epsilon \vDash \varphi$
   2) $au \vDash X\ \varphi \quad \rightsquigarrow \quad u \vDash \varphi$

5. Rules for $u \vDash \Diamond_t\ \varphi$:
   1) $\epsilon \vDash \Diamond_t\ \varphi \quad \rightsquigarrow \quad \epsilon \vDash \varphi$
   2) $u \vDash \Diamond_0\ \varphi \quad \rightsquigarrow \quad \bot$
   3) $u \vDash \Diamond_t\ \varphi \quad \rightsquigarrow \quad \top \quad$ if $u \vDash \varphi \rightsquigarrow^* \top$
   4) $au \vDash \Diamond_t\ \varphi \quad \rightsquigarrow \quad u \vDash \Diamond_{t-1}\ \varphi \quad$ if $au \vDash \varphi \rightsquigarrow^* \bot$

6. Rules for $u \vDash \Box_t\ \varphi$:
   1) $\epsilon \vDash \Box_t\ \varphi \quad \rightsquigarrow \quad \epsilon \vDash \varphi$
   2) $u \vDash \Box_0\ \varphi \quad \rightsquigarrow \quad \top$
   3) $u \vDash \Box_t\ \varphi \quad \rightsquigarrow \quad \bot \quad$ if $u \vDash \varphi \rightsquigarrow^* \bot$
   4) $au \vDash \Box_t\ \varphi \quad \rightsquigarrow \quad u \vDash \Box_{t-1}\ \varphi \quad$ if $au \vDash \varphi \rightsquigarrow^* \top$

7. Rules for $u \vDash \varphi_1\ U_t\ \varphi_2$:
   1) $\epsilon \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad \epsilon \vDash \varphi_2$
   2) $u \vDash \varphi_1\ U_0\ \varphi_2 \quad \rightsquigarrow \quad \bot$
   3) $u \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad \top \quad$ if $u \vDash \varphi_2 \rightsquigarrow^* \top$
   4) $u \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad \bot \quad$ if $u \vDash \varphi_1 \rightsquigarrow^* \bot \wedge u \vDash \varphi_2 \rightsquigarrow^* \bot$
   5) $au \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad u \vDash \varphi_1\ U_{t-1}\ \varphi_2 \quad$ if $au \vDash \varphi_1 \rightsquigarrow^* \top \wedge au \vDash \varphi_2 \rightsquigarrow^* \bot$

8. Rules for $u \vDash \varphi_1\ R_t\ \varphi_2$:
   1) $\epsilon \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad \epsilon \vDash \varphi_1$
   2) $u \vDash \varphi_1\ R_0\ \varphi_2 \quad \rightsquigarrow \quad \top$
   3) $u \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad \top \quad$ if $u \vDash \varphi_1 \rightsquigarrow^* \top \wedge u \vDash \varphi_2 \rightsquigarrow^* \top$
   4) $u \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad \bot \quad$ if $u \vDash \varphi_2 \rightsquigarrow^* \bot$
   5) $au \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad u \vDash \varphi_1\ R_{t-1}\ \varphi_2 \quad$ if $au \vDash \varphi_1 \rightsquigarrow^* \bot \wedge au \vDash \varphi_2 \rightsquigarrow^* \top$

for ϵ the empty word. These rules follow this schema: (i) an inconclusive value is returned when the empty word is found; (ii) the formula is appropriately evaluated when the timeout expires; (iii) it evaluates the subformulas to check whether a value can be obtained; it consumes the current letter and continues the evaluation; and (iv) inconclusive is returned if the subformulas are evaluated to inconclusive as well, and hence the previous rules cannot be applied. Hence, note that these rules have conditions that depend on the future. This happens in rules with a condition involving $\rightsquigarrow^*$ that inspects not only the first letter of the word, i.e., what is happening now, but also the subsequent letters, as illustrated by the following examples:

Example 4 *We recall the word* $u \equiv \boxed{(b,0)}\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)}$ *from Example 2 and evaluate the following formulas:*

- $\boxed{(b,0)}\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash \Box_2(\lambda^o_x.x = b) \rightarrow (\Diamond_2\ \lambda^p_y.y = a) \rightsquigarrow \bot$, *because first the* x *in* $(\lambda^o_x.x = b)$ *is bound to* b *and hence the premise holds, but* $\boxed{(b,0)}\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash (\Diamond_2 \lambda^p_y.y = a) \rightsquigarrow$ $\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash (\Diamond_1\ \lambda^p_y.y = a) \rightsquigarrow$ $\boxed{(a,3)}\boxed{(a,6)} \vDash (\Diamond_0 \lambda^p_y.y = a) \rightsquigarrow \bot$.

- $\boxed{(b,0)}\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash (\lambda^o_x.x = b)\ U_2\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a) \rightsquigarrow$ $\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash (\lambda^o_x.x = b)\ U_1\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)$, *which requires to check the second and third letters to check that the second formula does not hold. Then we have* $\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash (\lambda^o_x.x = b)\ U_1\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a) \rightsquigarrow \top$ *after checking the third and fourth letters.*

- $\boxed{(b,0)}\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash \lambda^o_x.\Box_{o+6}x = b \rightsquigarrow$ $\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash \Box_6\top$, *just by binding the variables. Then we have* $\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)} \vDash$

$\Box_6 \top \rightsquigarrow$
$\boxed{(a,3)}\ \boxed{(a,6)} \vDash \Box_5 \top \rightsquigarrow$
$\boxed{(a,6)} \vDash \Box_4 \top \rightsquigarrow \varepsilon \vDash \Box_3 \top \rightsquigarrow \varepsilon \vDash \top \rightsquigarrow \top$ *just by applying the rules for* $\Box$.

To use this procedure as the basis for our implementation, we would had to keep a list of suspended alternatives for each of the rules above, that are pending the resolution of the conditions that define each alternative, that will be solved in the future. For example if we apply rule 5 to an application of $\Diamond_t$ for a non empty word and $t > 0$ then we get 2 alternatives for sub-rules 5.3 and 5.4, and those alternatives will depend on whether the nested formula φ is reduced to $\top$ or $\bot$ using $\rightsquigarrow^*$, which cannot be determined at the instant when rule 5 is applied. This is because, although we do have all the batches for a generated test case corresponding to an input stream, the batches for output streams generated by transforming the input will be only generated after waiting the corresponding number of instants, as our implementation runs the actual code that is the subject of the test in a local Spark cluster. This leads to a complex and potentially expensive computation, since many pending possible alternatives have to be kept open. Instead of using this approach, it would be much more convenient to define a *stepwise* method with transition rules that only inspect the first letter of the input word.

2.2 A transformation for stepwise evaluation

In order to define this stepwise evaluation, it is worth noting that all the properties are finite (that is, all of them can be proved or disproved after a finite number of steps). It is hence possible to express any formula only using the temporal operators next and consume, which leads us to the following definition.

Definition 1 (Next form) *We say that a formula $\psi \in LTL_{ss}$ is in* next form *iff. it is built by using the following grammar:*

$$\psi ::= \quad \bot \mid \top \mid p(e,\ldots,e) \mid e = e \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi \rightarrow \psi \mid X\psi \mid \lambda^o_x.\psi$$

We can extend the transformation in [Riesco and Rodríguez-Hortalá, 2016b] for computing the next form of any formula $\varphi \in LTL_{ss}$:

Definition 2 (Explicit next transformation) *Given a formula $\varphi \in LTL_{ss}$, the function $nt^e(\varphi)$ computes another formula $\varphi' \in LTL_{ss}$, such that φ' is in* next form *and $\forall u \in \Sigma^*.u \vDash \varphi \iff u \vDash \varphi'$.*

$$\begin{array}{lcl}
nt^e(\top) & = & \top \\
nt^e(\bot) & = & \bot \\
nt^e(p(e_1,\ldots,e_n)) & = & p(e_1,\ldots,e_n) \\
nt^e(e_1 = e_2) & = & e_1 = e_2 \\
nt^e(\neg\varphi) & = & \neg nt^e(\varphi) \\
nt^e(\varphi_1 \vee \varphi_2) & = & nt^e(\varphi_1) \vee nt^e(\varphi_2) \\
nt^e(\varphi_1 \wedge \varphi_2) & = & nt^e(\varphi_1) \wedge nt^e(\varphi_2) \\
nt^e(\varphi_1 \rightarrow \varphi_2) & = & nt^e(\varphi_1) \rightarrow nt^e(\varphi_2) \\
nt^e(X\varphi) & = & X nt^e(\varphi) \\
nt^e(\lambda^o_x.\varphi) & = & \lambda^o_x.nt^e(\varphi) \\
nt^e(\Diamond_t\varphi) & = & nt^e(\varphi) \vee X nt^e(\varphi) \vee \ldots \vee X^{t-1} nt^e(\varphi) \\
nt^e(\Box_t\varphi) & = & nt^e(\varphi) \wedge X nt^e(\varphi) \wedge \ldots \wedge X^{t-1} nt^e(\varphi) \\
nt^e(\varphi_1\ U_t\ \varphi_2) & = & nt^e(\varphi_2) \vee (nt^e(\varphi_1) \wedge X nt^e(\varphi_2)) \vee \\
 & & (nt^e(\varphi_1) \wedge X nt^e(\varphi_1) \wedge X^2 nt^e(\varphi_2)) \vee \ldots \vee \\
 & & (nt^e(\varphi_1) \wedge X nt^e(\varphi_1) \wedge \ldots \wedge X^{t-2} nt^e(\varphi_1) \wedge X^{t-1} nt^e(\varphi_2)) \\
nt^e(\varphi_1\ R_t\ \varphi_2) & = & (nt^e(\varphi_2) \wedge X nt^e(\varphi_2) \wedge \ldots \wedge X^{t-1} nt^e(\varphi_2)) \vee \\
 & & (nt^e(\varphi_1) \wedge nt^e(\varphi_2)) \vee (nt^e(\varphi_2) \wedge X(nt^e(\varphi_1) \wedge nt^e(\varphi_2))) \vee \\
 & & (nt^e(\varphi_2) \wedge X nt^e(\varphi_2) \wedge X^2(nt^e(\varphi_1) \wedge nt^e(\varphi_2))) \vee \ldots \vee \\
 & & (nt^e(\varphi_2) \wedge X nt^e(\varphi_2) \wedge \ldots \wedge X^{t-2} nt^e(\varphi_2) \wedge X^{t-1}(nt^e(\varphi_1) \wedge nt^e(\varphi_2))
\end{array}$$

for $e_1, e_2 \in Term$, $x, o \in \mathcal{V}$, $p \in \mathcal{P}^n$, and $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$.

Note that (i) it is not always possible to compute the next form a priori, since the time in temporal operators might contain variables that need to be bound and (ii) the transformation might produce large formulas. For these reasons, it is worth transforming the formula following a lazy strategy, which only generates the subformulas required in the current and the next states. We present next a recursive function that allows us to compute the next form in a lazy way, which we use to improve the efficiency of our prototype, as we will see in Section 3.3:

Definition 3 (Recursive next transformation) *Given a formula $\varphi \in LTL_{ss}$, the function $nt(\varphi)$ computes another formula $\varphi' \in LTL_{ss}$, such that φ' is in* next form *and $\forall u \in \Sigma^*.u \vDash \varphi \iff u \vDash \varphi'$.*

$$
\begin{array}{lcll}
nt(\top) & = & \top & \\
nt(\bot) & = & \bot & \\
nt(p(e_1,\ldots,e_n)) & = & p(e_1,\ldots,e_n) & \\
nt(e_1 = e_2) & = & e_1 = e_2 & \\
nt(\neg\varphi) & = & \neg nt(\varphi) & \\
nt(\varphi_1 \vee \varphi_2) & = & nt(\varphi_1) \vee nt(\varphi_2) & \\
nt(\varphi_1 \wedge \varphi_2) & = & nt(\varphi_1) \wedge nt(\varphi_2) & \\
nt(\varphi_1 \rightarrow \varphi_2) & = & nt(\varphi_1) \rightarrow nt(\varphi_2) & \\
nt(X\varphi) & = & X\,nt(\varphi) & \\
nt(\lambda^o_x.\varphi) & = & \lambda^o_x.nt(\varphi) & \\
nt(\Diamond_1\varphi) & = & nt(\varphi) & \\
nt(\Diamond_t\varphi) & = & nt(\varphi) \vee X\,nt(\Diamond_{t-1}\varphi) & \text{if } t \geq 2 \\
nt(\Box_1\varphi) & = & nt(\varphi) & \\
nt(\Box_t\varphi) & = & nt(\varphi) \wedge X\,nt(\Box_{t-1}\varphi) & \text{if } t \geq 2 \\
nt(\varphi_1\ U_1\ \varphi_2) & = & nt(\varphi_2) & \\
nt(\varphi_1\ U_t\ \varphi_2) & = & nt(\varphi_2) \vee & \\
 & & (nt(\varphi_1) \wedge X\,nt(\varphi_1\ U_{t-1}\ \varphi_2)) & \text{if } t \geq 2 \\
nt(\varphi_1\ R_1\ \varphi_2) & = & nt(\varphi_1) \wedge nt(\varphi_2) & \\
nt(\varphi_1\ R_t\ \varphi_2) & = & (nt(\varphi_1) \wedge nt(\varphi_2)) \vee & \\
 & & (nt(\varphi_2) \wedge X\,nt(\varphi_1\ R_{t-1}\ \varphi_2)) & \text{if } t \geq 2
\end{array}
$$

for $e_1, e_2 \in Term$, $x, o \in \mathcal{V}$, $p \in \mathcal{P}^n$, and $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$.

It is easy to see that the relation $nt(\varphi)[x \mapsto v] \equiv nt(\varphi[x \mapsto v])$ holds, which will be useful in the implementation. Next, we present some results about these transformations:

Theorem 1 (Transformation equivalence) *Given a formula $\varphi \in LTL_{ss}$ such that φ does not contain variables in temporal connectives, we have $nt(\varphi) = nt^e(\varphi)$.*

It is straightforward to see that the formula obtained by this transformation is in *next form*, since it only introduces formulas using the temporal operators next or consume. The equivalence between formulas is stated in Theorem 2:

Theorem 2 *Given an alphabet Σ, an interpretation $\mathcal{A}$, and formulas $\varphi, \varphi' \in LTL_{ss}$, such that $\varphi' \equiv nt(\varphi)$, we have $\forall u \in (\Sigma \times \mathbb{N})^*.u \vDash^{\mathcal{A}} \varphi \iff u \vDash^{\mathcal{A}} \varphi'$.*

Both theorems are proved by induction in the structure of formulas and using an auxiliary lemma (see 1) that indicates that, if two formulas are equivalent at time 1, then they keep being equivalent after Detailed proofs are available in A.

Example 5 *We present here how to transform some of the formulas from Example 2. Note that the last one cannot be completely transformed a priori:*

- $nt(\Diamond_4\ \lambda^o_x.x = c) = \lambda^o_x.x = c \vee X\lambda^o_x.x = c \vee X^2\lambda^o_x.x = c \vee X^3\lambda^o_x.x = c$
- $nt(\Box_4\ \lambda^o_x.(x = a \vee x = b)) = (x = a \vee x = b) \wedge X(x = a \vee x = b) \wedge$
  $X^2(x = a \vee x = b) \wedge X^3(x = a \vee x = b)$
- $nt(\lambda^o_x.x = b\ U_2\ \lambda^o_x.x = a) = \lambda^o_x.x = a \vee (\lambda^o_x.x = b \wedge X\lambda^o_x.x = a)$
- $nt(\lambda^o_x.x = a\ R_2\ \lambda^p_y.y = b) = (\lambda^o_x.x = a \wedge \lambda^p_y.y = b) \vee (\lambda^o_x.x = a \wedge X(\lambda^o_x.x = a \wedge \lambda^p_y.y = b))$
- $nt(\Box_2(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a) = (\lambda^o_x.x = b \rightarrow (\lambda^p_y.y = a \vee X\lambda^p_y.y = a)) \wedge X(\lambda^o_x.x = b \rightarrow (\lambda^p_y.y =$
  $a \vee X\lambda^p_y.y = a))$
- $nt((\lambda^o_x.x = b)\ U_2\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)) = X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a) \vee (\lambda^o_x.x = b \wedge X^2(\lambda^p_y.y =$
  $a \wedge X\lambda^q_z.z = a))$

The following example illustrates this lazy behavior:

Example 6 *We present the lazy next transformation for some formulas, where we just apply the first transformation. Note that in the last example it is not possible to compute the next form in an eager way:*

- $nt(\Box_2(\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a)) = (\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a) \wedge Xnt(\Box_1(\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a))$
- $nt(\lambda^o_x.\Box_{o+6}x = b) = \lambda^o_x.nt(\Box_{o+6}x = b)$

Once the next form of a formula has been computed, it is possible to evaluate it for a given word just by traversing its letters. We just evaluate the atomic formulas in the present moment (that is, those properties that does not contain the next operator) and remove the next operator otherwise, so these properties will be evaluated for the next letter. This method is detailed as follows:

Definition 4 (Letter simplification) *Given a formula ψ in next form, a letter $s \in \Sigma$, where s can be either (e, t), with $e \in Term, t \in \mathbb{N}$, or the empty letter $\emptyset$, and an interpretation $\mathcal{A} = (A, I)$, the function $ls^{\mathcal{A}}(\psi, s)$ ($ls(\psi, s)$ when $\mathcal{A}$ is clear from the context)* simplifies ψ with s *as follows:*

- $ls(\top, s) = \top$.
- $ls(\bot, s) = \bot$.
- $ls^{\mathcal{A}}(p(e_1, \ldots, e_n), s) = (\llbracket e_1 \rrbracket^{\mathcal{A}}, \ldots, \llbracket e_n \rrbracket^{\mathcal{A}}) \subseteq I(p)$.
- $ls^{\mathcal{A}}(e_1 = e_2, s) = synEq(\llbracket e_1 \rrbracket^{\mathcal{A}}, \llbracket e_2 \rrbracket^{\mathcal{A}})$.
- $ls(\psi_1 \vee \psi_2, s) = ls(\psi_1, s) \vee ls(\psi_2, s)$.
- $ls(\psi_1 \wedge \psi_2, s) = ls(\psi_1, s) \wedge ls(\psi_2, s)$.
- $ls(\psi_1 \to \psi_2, s) = ls(\psi_1, s) \to ls(\psi_2, s)$.
- $ls(X\psi, (e, t)) = \psi$.
- $ls(X\psi, \emptyset) = ls(\psi, \emptyset)$.
- $ls(\lambda^o_x.\psi, (e, t)) = \psi[x \mapsto e][o \mapsto t]$.
- $ls(\lambda^o_x.\psi, \emptyset) = ?$.

where *synEq* stands for syntactic equality. Note that using the empty letter forces the complete evaluation of the formula. Using this function and applying propositional logic and the interpretation $\mathcal{A}$ when definite values are found it is possible to evaluate formulas in a step-by-step fashion.[7] In this way, we can solve the formulas from the previous example as follows:

Example 7 *We present here the lazy evaluation process for some formulas in Example 4 using the word* $u \equiv \boxed{(b,0)}\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)}$.

- $nt(\Box_2(\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a)) = (\lambda^o_x.x = b) \to (\lambda^p_y.y = a \vee (Xnt(\Diamond_1\lambda^p_y.y = a))) \wedge Xnt(\Box_1(\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a))$ *(from Example 6).*
  - $ls((\lambda^o_x.x = b) \to (\lambda^p_y.y = a \vee (Xnt(\Diamond_1\lambda^p_y.y = a))) \wedge Xnt(\Box_1(\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a)), (b, 0))) =$ *(consume letter)*
    $\top \to (\bot \vee nt(\Diamond_1\lambda^p_y.y = a)) \wedge nt(\Box_1(\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a))$
    $\equiv$ *(simplification)*
    $nt(\Diamond_1\lambda^p_y.y = a) \wedge nt(\Box_1(\lambda^o_x.x = b) \to (\Diamond_2\lambda^p_y.y = a))$
    $\equiv$ *(lazy evaluation of the next transformation)*
    $\lambda^p_y.y = a \wedge \lambda^o_x.x = b \to (\lambda^p_y.y = a \vee Xnt(\Diamond_1\lambda^p_y.y = a)))$.
  - $ls(\lambda^p_y.y = a \wedge \lambda^o_x.x = b \to (\lambda^p_y.y = a \vee Xnt(\Diamond_1\lambda^p_y.y = a))), (b, 2))$
    $=$ *(consume letter)*
    $\bot \wedge \top \to (\bot \vee nt(\Diamond_1\lambda^p_y.y = a)))$
    $\equiv$ *(simplification)*
    $\bot$.

[7] Note that the value ? is only reached when the word is consumed and this simplification cannot be applied.

- $nt(\lambda^o_x.\Box_{o+6}x = b) = \lambda^o_x.nt(\Box_{o+6}x = b)$ *(from Example 6).*
  - $ls(\lambda^o_x.nt(\Box_{o+6}x = b), (b, 0)) = nt(\Box_6 b = b)$
    $\equiv$ *(lazy evaluation of the next transformation)*
    $\top \wedge X\, nt(\Box_5 b = b)$
    $\equiv$ *(simplification)*
    $X\, nt(\Box_5 b = b)$
  - $ls(X\, nt(\Box_5 b = b), (b, 2)) = nt(\Box_5 b = b)$
    $\equiv$ *(lazy evaluation of the next transformation and simplification)*
    $X\, nt(\Box_4 b = b)$
  - $ls(X\, nt(\Box_4 b = b), (a, 3)) = nt(\Box_4 b = b)$
    $\equiv$ *(lazy evaluation of the next transformation and simplification)*
    $X\, nt(\Box_3 b = b)$
  - $ls(X\, nt(\Box_3 b = b), (a, 6)) = nt(\Box_3 b = b)$
    $\equiv$ *(lazy evaluation of the next transformation and simplification)*
    $X\, nt(\Box_2 b = b)$
  - $ls(X\, nt(\Box_3 b = b), \emptyset) = nt(\Box_3 b = b)$
    $\equiv$ *(lazy evaluation of the next transformation and simplification)*
    $X\, nt(\Box_2 b = b)$
  - $ls(X\, nt(\Box_2 b = b), \emptyset) = ls(nt(\Box_2 b = b), \emptyset)$
    $\equiv$ *(lazy evaluation of the next transformation and simplification)*
    $ls(nt(\Box_1 b = b), \emptyset)$
    $\equiv$ *(lazy evaluation of the next transformation and simplification)*
    $ls(\top, \emptyset) \equiv \top$

When no variables appear in the timeouts of temporal operators, the next transformation gives also the intuition that inconclusive values can be avoided if we use a word as long as the number of next/consume operators nested in the transformation.[8] We define this *safe word length* as follows:

Definition 5 (Safe word length) *Given a formula $\varphi \in LTL_{ss}$ without variables in any timeouts of the temporal operators that occur in it, its longest required check $swl(\varphi) \in \mathbb{N}$ is the maximum word length of a word u such that we have $u \vDash \varphi \in \{\top, \bot\}$. It is defined as follows:*

$$
\begin{array}{lcl}
swl(\top) & = & 0 \\
swl(\bot) & = & 0 \\
swl(f(e_1, \ldots, e_n)) & = & 0 \\
swl(t_1 = t_2) & = & 0 \\
swl(\neg\varphi) & = & swl(\varphi) \\
swl(\varphi_1 \vee \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) \\
swl(\varphi_1 \wedge \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) \\
swl(\varphi_1 \rightarrow \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) \\
swl(X\varphi) & = & swl(\varphi) + 1 \\
swl(\Diamond_t\varphi) & = & swl(\varphi) + (t - 1) \\
swl(\Box_t\varphi) & = & swl(\varphi) + (t - 1) \\
swl(\varphi_1\ U_t\ \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) + (t - 1) \\
swl(\varphi_1\ R_t\ \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) + (t - 1) \\
swl(\lambda^o_x.\varphi) & = & swl(\varphi) + 1
\end{array}
$$

Example 8 *We present here the safe word length for some of the formulas in Example 2:*

- $swl(\Diamond_4\ \lambda^o_x.x = c) = 4$.
- $swl(\lambda^o_x.x = b\ U_2\ \lambda^p_y.y = a) = 2$.
- $swl(\Box_3(\lambda^o_x.x = a) \rightarrow X(\lambda^p_y.y = a)) = 4$.
- $swl(\Box_2(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a)) = 3$.
- $swl((\lambda^o_x.x = b)\ U_2\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)) = 4$.

On the other hand, we cannot define a safe word length for arbitrary formulas with variables in timeouts, because an application of the consume operator might bind those variables using a letter of the input word, so there is no way to determine the value of the timeout for all possible words.

[8] Note that it might be possible to avoid an inconclusive value with shorter words, so this is a *sufficient* condition.

2.3 Generating words

Besides stating properties, formulas can be used to generate words. In particular, we will generate sequences of terms from formulas; these sequences can then be extended by pairing each letter with a number generated by an arbitrary monotonically increasing function, hence obtaining words with timed terms as letters. The formulas used for generating terms have the following restrictions:

- Given a formula $\lambda_x^o.\varphi$, we have $o \notin fv(\varphi)$. Since in this stage we do not generate times, they cannot be used.
- Formulas do not contain the negation operator or the `false` constant. The process tries to generate words that make the formula evaluate to true, so we would not generate any word for a contradiction. Besides, we do not support negation because that would imply maintaining a set of excluded words, and we wanted to define simple ScalaCheck generators in the most straightforward way.

For describing how the generators compute the sequences we first need to introduce a constant $err \in Term^*$ that stands for an erroneous sequence. Moreover, we use the notation $+ : Term^* \times Term^* \to Term^*$ $(u + err = err + u = err)$ for composing words, and extend the union on $Term^*$ as:

$$\begin{array}{lcl} a\ u \cup b\ v & = & (a \cup b) + u \cup v \\ u \cup \epsilon & = & u \\ u \cup err & = & err \end{array}$$

for $a, b \in Term$ and $u, v \in Term^*$. Note that we assume that syntax for sets and unions is defined in $\mathcal{F}$. Using these ideas, we have:

Definition 6 (Random word generation) *Given an interpretation $\mathcal{A}$, $e_1, \ldots, e_n \in Term$, $p \in \mathcal{P}^n$, formulas ψ, ψ_1, and ψ_2 in next form, the function $gen^{\mathcal{A}}$ (gen when $\mathcal{A}$ is clear from the context) generates a finite word $u \in Term^*$. If different equations can be applied for a given formula any of them can be chosen:*

$gen(\top)$	$=$	$\emptyset$	
$gen^{\mathcal{A}}(p(e_1, \ldots, e_n))$	$=$	$\emptyset$	*if* $([\![e_1]\!]^{\mathcal{A}}, \ldots, [\![e_n]\!]^{\mathcal{A}}) \subseteq I(p)$
$gen^{\mathcal{A}}(e_1 = e_2)$	$=$	$\emptyset$	*if* $[\![e_1]\!]^{\mathcal{A}} = [\![e_2]\!]^{\mathcal{A}}$
$gen(\psi_1 \vee \psi_2)$	$=$	$gen(\psi_1)$	
$gen(\psi_1 \vee \psi_2)$	$=$	$gen(\psi_2)$	
$gen(\psi_1 \wedge \psi_2)$	$=$	$gen(\psi_1) \cup gen(\psi_2)$	
$gen(\psi_1 \to \psi_2)$	$=$	$gen(\psi_2)$	
$gen(X\psi)$	$=$	$\emptyset + gen(\psi)$	
$gen(\lambda_x^o.\psi)$	$=$	$\{e\} + gen(\psi)$	*if* $x \notin fv(\psi)$, *pick an* $e \in Term$ *s.t.* $gen(\psi[x \mapsto e]) \neq err$
$gen(\psi)$	$=$	err	otherwise

where $\emptyset$ stands for an empty term and indicates that the batch can be empty.

Note that this definition interprets conjunctions as unions. Hence, the formula $\psi \equiv (\lambda_x^o.x = a) \wedge (\lambda_x^o.x = b)$ is interpreted as $\psi \equiv (\lambda_x^o.x \supset \{a\}) \wedge (\lambda_x^o.x \supset \{b\})$ and generates a single batch containing a and b.

Example 9 *We present here the generation process for a formula from Example 2.*

- $gen(\Box_2(\lambda_x^o.x = b) \to (\Diamond_2 \lambda_y^p.y = a)) = gen((\lambda_x^o.x = b) \to (\lambda_y^p.y = a \vee (Xnt(\Diamond_1 \lambda_y^p.y = a))) \wedge Xnt(\Box_1(\lambda_x^o.x = b) \to (\Diamond_2 \lambda_y^p.y = a)))$ *(from Example 6).*
- $gen((\lambda_x^o.x = b) \to (\lambda_y^p.y = a \vee (Xnt(\Diamond_1 \lambda_y^p.y = a))) \wedge Xnt(\Box_1(\lambda_x^o.x = b) \to (\Diamond_2 \lambda_y^p.y = a))) =$
  $gen((\lambda_x^o.x = b) \to (\lambda_y^p.y = a \vee (Xnt(\Diamond_1 \lambda_y^p.y = a)))) \cup$
  $gen(Xnt(\Box_1(\lambda_x^o.x = b) \to (\Diamond_2 \lambda_y^p.y = a))) =$
  $\boxed{a} \cup \boxed{\emptyset}\ \boxed{a} = \boxed{a}\ \boxed{a}$

Since we have, for the first term of the union:

- $gen((\lambda_x^o.x = b) \to (\lambda_y^p.y = a \vee (Xnt(\Diamond_1 \lambda_y^p.y = a)))) =$
  $gen(\lambda_y^p.y = a \vee (Xnt(\Diamond_1 \lambda_y^p.y = a))) =$
  $gen(\lambda_y^p.y = a) =$
  $\boxed{a}$

Similarly we would generate the second term of the union. Note that in both cases we decided to generate values for the first term of the disjunction. A similar process can be followed to obtain different values.

3 sscheck: using LTL_{ss} for property-based testing

We have developed a prototype that allows using the LTL_{ss} logic for property-based testing of Spark Streaming programs, as the Scala library sscheck [Riesco and Rodríguez-Hortalá, 2017]. This library extends the popular PBT library ScalaCheck [Nilsson, 2014] with custom generators for Spark DStreams, and with a property factory that allows developers to check a LTL_{ss} formula against the finite DStream prefixes generated by another LTL_{ss} formula.

3.1 Architecture overview

In order to write a temporal property in sscheck, the user extends the trait (the Scala version of an abstract class) `DStreamTLProperty`, and then implements some abstract methods to configure Spark Streaming (e.g. defining the batch interval or the Spark master). The method `DStreamTLProperty.forAllDStream` is used to define temporal ScalaCheck properties:

```
type SSeq[A] = Seq[Seq[A]]
type SSGen[A] = Gen[SSeq[A]]

def forAllDStream[In:ClassTag,Out:ClassTag](
  generator: SSGen[In])(
  transformation: (DStream[In]) => DStream[Out])(
  formula: Formula[(RDD[In], RDD[Out])])(
  implicit pp1: SSeq[In] => Pretty): Prop
```

The function `forAllDStream` takes a ScalaCheck generator of sequences of sequences of elements, that are interpreted as finite DStream prefixes, so each nested sequence is interpreted as an RDD. Our library defines a case class `Batch[A]` that extends `Seq[A]` to represent an RDD for a micro batch, and a case class `PDStream[A]` that extends `Seq[Batch[A]]` to represent a finite DStream prefix. For example `Batch("scala", "spark")` represents an `RDD[String]` with 2 elements, and `PDStream(Batch("scala", "spark"), Batch(), Batch("spark"))` represents a finite prefix of a `DStream[String]` consisting in a micro batch with 2 elements, followed by an empty micro batch, and finally a micro batch with a single element. The sscheck classes `BatchGen` and `PDStreamGen` and their companion objects can be used to define generators of `Batch` and `PDStream` objects—hence generators of `SSeq[A]`—using temporal operators, that are interpreted according to the LTL_{ss} logic as described in Section 2.3. The trait `Formula` is used to represent LTL_{ss} formulas. This trait is extended in different case classes `Always`, `And`, etc. corresponding to the different logical operators, thus representing the set of formulas LTL_{ss} as an algebraic data type, as usual in functional programming. All basic literals corresponding to formulas of the shape of $\bot$, $\top$, $p(e_1, \ldots, e_n)$, and $e_1 = e_2$ are represented by the case class `Solved`, that takes an expression of type `org.scalacheck.Prop.Status` or `org.specs2.execute.Result` for the literal. Note the type parameter of `Formula` is `(RDD[In], RDD[Out])`, which means in `formula` the letter corresponding to each instant is a pair of RDDs, one for the input DStream and another for the output DStream. Finally the function `transformation` is the *test subject* which correctness is checked during the evaluation of the property.

In order to evaluate the resulting ScalaCheck `Prop`, first we apply a lazy variant of the transformation from Definition 3 (see Section 3.3 for details.) to `formula`, in order to get an equivalent formula in next form. Then the following process iterates until the specified number of test cases has passed, or until a failing test case—i.e. a counterexample—is found, whatever happens first. A test case of type `SSeq[In]` is generated using `generator`, which corresponds to a finite prefix for the input DStream, and a fresh Spark `StreamingContext` is created. The test case, the streaming context, and the transformation are used to create a `TestCaseContext` that encapsulates the execution of the test case. The program then blocks until the test case is executed completely by the Spark runtime, and then a result for the test case is returned by the test case context. Test case results can be inconclusive, which corresponds to the truth value ? in LTL_{ss}, in case the generated test case is too short for the formula. Internally the test case context defines an input DStream by parallelizing the test case —using the Spark-testing-base package [Holden Karau, 2015a]—, and applies the test subject `transformation` to it to define an output

DStream. It also maintains variables for the number of remaining batches (initialized to the length of the test case), and the current value for the formula, and registers a `foreachRDD` Spark action that updates the number of remaining batches, and the current formula using the letter simplification procedure from Definition 4. This action also stops the Spark streaming context once the formula is solved or there are no remaining batches. Other variants of `forAllDStream` can be used for defining properties with more than one input DStream and one output DStream.

Therefore `forAllDStream(gen)(transformation)(formula)` is trying to prove $\forall u_g \in gen(\varphi_g).(u, 1 \vDash^{\mathcal{A}^{\mathcal{S}}} \varphi_p : \top) \vee (u, 1 \vDash^{\mathcal{A}^{\mathcal{S}}} \varphi_p : ?)$ for the Spark interpretation structure $\mathcal{A}^{\mathcal{S}}$, formulas φ_g, φ_p corresponding to `gen` and `formula` respectively, and $u \equiv zip(zip(u_g, u_o), u_t)$ where u_o is a word which interpretation under $\mathcal{A}^{\mathcal{S}}$ corresponds to the result of applying `transformation` to the interpretation of u_g under $\mathcal{A}^{\mathcal{S}}$, and $u_t = ct\ (ct + b)\ (ct + 2b)\ (ct + 3b) \ldots$ is the sequence of time stamps starting from the unix timestamp ct at the start of the execution of the property and moving b milliseconds at a time for b the configured batch interval. Here zip is the usual operator that combines two sequences element wise to produce a sequence of pairs of elements in the same position, truncating the longest of the two sequences to the length of the shortest. This way we add an additional external universal quantifier on the domain of finite words, as usual in PBT, but inside that scope we have a propositional LTL_{ss} formula, and we evaluate the whole formula with the usual sound but incomplete PBT evaluation procedure.

Example 10 *Let's ground all those ideas with the following simple property, that checks that if we generate a stream of positive integers, and we filter out the negative numbers, then we get a stream of numbers that are greater or equal to zero.*

```
class SimpleStreamingFormulas
  extends org.specs2.Specification
  with DStreamTLProperty
  with org.specs2.ScalaCheck {

  // Spark configuration
    // run in local mode using 1 worker per machine core
  override def sparkMaster : String = "local[*]"
    // use batch interval of 100 milliseconds
  override def batchDuration = Duration(100)
   // use 4 partitions by default when creating
   // DStreams for test cases
  override def defaultParallelism = 4

  def is =
    sequential ^ s2"""
    Simple demo Sscheck example for ScalaCheck
      - Given a stream of integers
        When we filter out negative numbers
        Then we get only numbers greater or equal to
          zero $filterOutNegativeGetGeqZero
      """

    def filterOutNegativeGetGeqZero = {
      type U = (RDD[Int], RDD[Int])
      val numBatches = 10
      val gen = BatchGen.always(BatchGen.ofNtoM(10, 50, arbitrary[Int]),
                                numBatches)
      val formula = always(nowTime[U]{ (letter, time) =>
        val (_input, output) = letter
        output should foreachRecord {_ >= 0}
      }) during numBatches

      forAllDStream(
      gen)(
      _.filter{ x => !(x < 0)})(
      formula)
  }.set(minTestsOk = 50).verbose
}
```

Here we use the type alias `U` for the letters. Also here we use the testing library Specs2 [Torreborre, 2014] to handle the creation and closing of the Spark contexts, as part of the implementation of `DStreamTLProperty`. Besides, we have defined some custom Specs2 `Matchers` on RDDs that are helpful when defining properties—see Appendix ?? for details.

```
/** @return a matcher that checks whether predicate holds for all the
 *          records of an RDD or not
 */
def foreachRecord[T](predicate: T => Boolean): Matcher[RDD[T]]
/** @return a matcher that checks whether predicate holds for at least
 *          one of the records of an RDD or not.
 */
def existsRecord[T](predicate: T => Boolean): Matcher[RDD[T]]
/** @return a Matcher that checks that both RDDs are equal as sets
 */
def beEqualAsSetTo[T](actualRDD: RDD[T]): Matcher[RDD[T]]
```

3.2 Verifying AMP Camp's Twitter tutorial with sscheck

Now we will present a more complex example, adapted for Berkeley's AMP Camp training on Spark,[9] but adding sscheck properties for the functions implemented in that tutorial. The complete code for these examples is available at `https://github.com/juanrh/sscheck-examples/releases/tag/0.0.4`.

Our test subject will be an object `TweetOps`—see Appendix ?? for the source code—that defines a series of operations on a stream of tweets. A tweet is a piece of text of up to 140 characters, together with some meta-information like an identifier for the author or the creation date. Those words in a tweet that start with the `#` character are called "hashtags", and are used by the tweet author to label the tweet, so other users that later search for tweets with a particular hashtag might locate those related tweets easily. The operations below take a stream of tweets and, respectively, generate the stream for the set of hashtags in all the tweets; the stream of pairs (hashtags, number of occurrences) in a sliding time window with the specified size[10]; and the stream that contains a single element for the most popular hashtag, i.e. the hastag with the highest number of occurrences, again for the specified time window.

```
object TweetOps {
 def getHashtags(tweets: DStream[Status]): DStream[String]
 def countHashtags(batchInterval: Duration, windowSize: Int)
                  (tweets: DStream[Status]): DStream[(String, Int)]
 def getTopHashtag(batchInterval: Duration, windowSize: Int)
                  (tweets: DStream[Status]): DStream[String]
}
```

In this code, the class `twitter4j.Status` from the library Twitter4J [Yamamoto, 2010] is used to represent each particular tweet. In the original AMP Camp training, the class `TwitterUtils`[11] is used to define a `DStream[Status]` by repeatedly calling the Twitter public API to ask for new tweets. Instead, in this example we replace the Twitter API by an input DStream defined by using an sscheck generator, so we can control the shape of the tweets that will be used as the test inputs. To do that we employ the mocking [Mackinnon et al., 2001] library Mockito [Kaczanowski, 2012] for stubbing [Fowler, 2007] `Status` objects, i.e. to easily synthetize objects that impersonate a real `Status` object, and that provide predefined answers to some methods, in this case the method that returns the text for a tweet. The different functions of the `TwitterGen` object produce such DStreams of mock `Status` objects —see Appendix ?? for details.

```
object TwitterGen {
  /** Generator of Status mocks with a getText method
   *  that returns texts of up to 140 characters
   *
   *  @param noHashtags if true then no hashtags are generated in the
   *  tweet text
   * */
```

[9] `http://ampcamp.berkeley.edu/3/exercises/realtime-processing-with-spark-streaming.html`

[10] See `https://spark.apache.org/docs/1.6.2/streaming-programming-guide.html#window-operations` for details on Spark Streaming window operators.

[11] `https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/streaming/twitter/TwitterUtils.html`

```
  def tweet(noHashtags: Boolean = true): Gen[Status]
  /** Take a Status mocks generator and return a Status mocks
   *  generator that adds the specified hashtag to getText
   * */
  def addHashtag(hashtagGen: Gen[String])
                (tweetGen: Gen[Status]): Gen[Status]
  def tweetWithHashtags(possibleHashTags: Seq[String]): Gen[Status]
  def hashtag(maxLen: Int): Gen[String]
  def tweetWithHashtagsOfMaxLen(maxHashtagLength: Int): Gen[Status]
}
```

3.2.1 Extracting hashtags

Now we are ready to write our first property, which checks that `getHashtags` works correctly, that is, it computes the set of *hashtags* (words starting with `#`). In the property we generate tweets that use a predefined set of hashtags, and then we check that all hashtags produced in the output are contained in that set.

```
def getHashtagsOk = {
  type U = (RDD[Status], RDD[String])
  val hashtagBatch = (_ : U)._2

  val numBatches = 5
  val possibleHashTags = List("#spark", "#scala", "#scalacheck")
  val tweets = BatchGen.ofNtoM(5, 10,
                               tweetWithHashtags(possibleHashTags)
                               )
  val gen = BatchGen.always(tweets, numBatches)

  val formula = always {
    at(hashtagBatch){ hashtags =>
      hashtags.count > 0 and
      ( hashtags should foreachRecord(possibleHashTags.contains(_)) )
    }
  } during numBatches

  forAllDStream(
    gen)(
    TweetOps.getHashtags)(
    formula)
}
```

This is a very basic safety property. Note also the different style we have employed for accessing the different RDDs for the current letter. While in the property from Example 10 we used matching with a partial function to access the batch for the output DStream, here we use `Function.at`, that allows us to combine a projection function with an assertion that uses the result of the projection.

In the next example we use the "reference implementation" PBT technique [Nilsson, 2014] to check the implementation of `TweetOps.getHashtags`, which is based on the Spark transformations `flatMap` and `filter` also using `String.startsWith`, against a regexp-based reference implementation. This gives us a more thorough test, because we use a different randomly generated set of hashtags for each batch of each test case, instead of a predefined set of hashtags for all the test cases.

```
private val hashtagRe = """#\S+""".r
private def getExpectedHashtagsForStatuses(statuses: RDD[Status])
: RDD[String] =
  statuses.flatMap { status => hashtagRe.findAllIn(status.getText)}

def getHashtagsReferenceImplementationOk = {
  type U = (RDD[Status], RDD[String])
  val (numBatches, maxHashtagLength) = (5, 8)

  val tweets = BatchGen.ofNtoM(5, 10,
                               tweetWithHashtagsOfMaxLen(maxHashtagLength))
```

```
  val gen = BatchGen.always(tweets, numBatches)

  val formula = alwaysR[U] { case (statuses, hashtags) =>
    val expectedHashtags = getExpectedHashtagsForStatuses(statuses).cache()
    hashtags must beEqualAsSetTo(expectedHashtags)
  } during numBatches

  forAllDStream(
    gen)(
    TweetOps.getHashtags)(
    formula)
}
```

3.2.2 Counting hashtags

In order to check `countHashtags`, in the following property we setup a scenario where the hashtag `#spark` is generated for some period, and then the hashtag `#scala` is generated for another period, and we express the expected counting behaviour with several subformulas: we expect to get the expected count of hashtags for spark for the first period (`laterAlwaysAllSparkCount`); we expect to eventually get the expected count of hastags for scala (`laterScalaCount`); and we expect that after reaching the expected count for spark hashtags, we would then decrease the count as time passes and elements leave the sliding window (`laterSparkCountUntilDownToZero`).

```
def countHashtagsOk = {
  type U = (RDD[Status], RDD[(String, Int)])
  val countBatch = (_ : U)._2

  val windowSize = 3
  val (sparkTimeout, scalaTimeout) = (windowSize * 4, windowSize * 2)
  val sparkTweet = tweetWithHashtags(List("#spark"))
  val scalaTweet = tweetWithHashtags(List("#scala"))
  val (sparkBatchSize, scalaBatchSize) = (2, 1)
  val gen = BatchGen.always(BatchGen.ofN(sparkBatchSize, sparkTweet),
                                        sparkTimeout) ++
            BatchGen.always(BatchGen.ofN(scalaBatchSize, scalaTweet),
                                        scalaTimeout)

  def countNHashtags(hashtag : String)(n : Int)  =
    at(countBatch)(_ should existsRecord(_ == (hashtag, n : Int)))
  val countNSparks = countNHashtags("#spark") _
  val countNScalas = countNHashtags("#scala") _
  val laterAlwaysAllSparkCount =
    later {
        always {
          countNSparks(sparkBatchSize * windowSize)
        } during (sparkTimeout -2)
    } on (windowSize + 1)
  val laterScalaCount =
    later {
      countNScalas(scalaBatchSize * windowSize)
    } on (sparkTimeout + windowSize + 1)
  val laterSparkCountUntilDownToZero =
    later {
      { countNSparks(sparkBatchSize * windowSize) } until {
        countNSparks(sparkBatchSize * (windowSize - 1)) and
          next(countNSparks(sparkBatchSize * (windowSize - 2))) and
          next(next(countNSparks(sparkBatchSize * (windowSize - 3))))
        } on (sparkTimeout -2)
    } on (windowSize + 1)
  val formula =
    laterAlwaysAllSparkCount and
    laterScalaCount and
    laterSparkCountUntilDownToZero
```

```
  forAllDStream(
    gen)(
    TweetOps.countHashtags(batchInterval, windowSize)(_))(
    formula)
}
```

Then we check the safety of `countHashtags` by asserting that any arbitrary generated hashtag is never skipped in the count. Here we again exploit the reference implementation technique to extract the expected hashtags, and join this with the output counts, so we can assert that all and only all expected hastags are counted, and that those countings are never zero at the time the hashtag is generated.

```
def hashtagsAreAlwasysCounted = {
  type U = (RDD[Status], RDD[(String, Int)])
  val windowSize = 3
  val (numBatches, maxHashtagLength) = (windowSize * 6, 8)

  val tweets = BatchGen.ofNtoM(5, 10,
                               tweetWithHashtagsOfMaxLen(maxHashtagLength))
  val gen = BatchGen.always(tweets, numBatches)

  val alwaysCounted = alwaysR[U] { case (statuses, counts) =>
    val expectedHashtags = getExpectedHashtagsForStatuses(statuses).cache()
    val expectedHashtagsWithActualCount =
      expectedHashtags
       .map((_, ()))
       .join(counts)
       .map{case (hashtag, (_, count)) => (hashtag, count)}
       .cache()
    val countedHashtags = expectedHashtagsWithActualCount.map{_._1}
    val countings = expectedHashtagsWithActualCount.map{_._2}

    // all hashtags have been counted
    countedHashtags must beEqualAsSetTo(expectedHashtags) and
    // no count is zero
    (countings should foreachRecord { _ > 0 })
  } during numBatches

  forAllDStream(
    gen)(
    TweetOps.countHashtags(batchInterval, windowSize)(_))(
    alwaysCounted)

}
```

3.2.3 Getting the most popular hashtag

Now we check the correctness of `getTopHashtag`, that extracts the most "popular" hashtag, i.e. the hashtag with the highest number of occurrences at each time window. For that we use the following property where we define a scenario in which we start with the hashtag `#spark` as the most popular (generator `sparkPopular`), and after that the hashtag `#scala` becomes the most popular (generator `scalaPopular`), and asserting on the output DStream that `#spark` is the most popular hashtag until `#scala` is the most popular.

```
def sparkTopUntilScalaTop = {
  type U = (RDD[Status], RDD[String])

  val windowSize = 1
  val topHashtagBatch = (_ : U)._2
  val scalaTimeout = 6
  val sparkPopular =
    BatchGen.ofN(5, tweetWithHashtags(List("#spark"))) +
    BatchGen.ofN(2, tweetWithHashtags(List("#scalacheck")))
```

```
  val scalaPopular =
    BatchGen.ofN(7, tweetWithHashtags(List("#scala"))) +
    BatchGen.ofN(2, tweetWithHashtags(List("#scalacheck")))
  val gen = BatchGen.until(sparkPopular, scalaPopular, scalaTimeout)

  val formula =
    { at(topHashtagBatch)(_ should foreachRecord(_ == "#spark" )) } until {
      at(topHashtagBatch)(_ should foreachRecord(_ == "#scala" ))
    } on (scalaTimeout)

  forAllDStream(
    gen)(
    TweetOps.getTopHashtag(batchInterval, windowSize)(_))(
    formula)
}
```

Finally, we state the safety of `getTopHastag` by checking that there is always one top hashtag.

```
def alwaysOnlyOneTopHashtag = {
  type U = (RDD[Status], RDD[String])
  val topHashtagBatch = (_ : U)._2

  val (numBatches, maxHashtagLength) = (5, 8)
  val tweets =
    BatchGen.ofNtoM(5, 10,
                    tweetWithHashtagsOfMaxLen(maxHashtagLength))

  val gen = BatchGen.always(tweets, numBatches)
  val formula = always {
    at(topHashtagBatch){ hashtags =>
      hashtags.count === 1
    }
  } during numBatches

  forAllDStream(gen)(
    TweetOps.getTopHashtag(batchInterval, 2)(_))(
    formula)
}
```

3.2.4 Defining liveness properties with the consume operator

So far we have basically defined two types of properties: properties where we simulate a particular scenario, and safety properties where we assert that we will never reach a particular "bad" state. It would be also nice to be able to write liveness properties in sscheck, which is another class of properties typically used with temporal logic, where we express that something good keeps happening with a formula of the shape of $\square_{t_1}(\varphi_1 \rightarrow \lozenge_{t_2}\varphi_2)$. In this kind of formulas it would be useful to define the conclusion formula φ_2 that should happen later, based on the value of the word that happened when the premise formula φ_1 was evaluated. This was our motivation for adding to the LTL_{ss} logic the consume operator $\lambda^o_x.\varphi$, that can be used in liveness formulas of the shape $\square_{t_1}(\lambda^o_x.\lozenge_{t_2}\varphi_2)$ or $\square_{t_1}(\lambda^o_x.\varphi_1 \rightarrow \lozenge_{t_2}\varphi_2)$. One example of the former is the following liveness property for `countHashtags`, that checks that always each hashtag eventually gets a count of 0, if we generate empty batches at the end of the test case so all hashtags end up getting out of the counting window.

```
def alwaysEventuallyZeroCount = {
  type U = (RDD[Status], RDD[(String, Int)])
  val windowSize = 4
  val (numBatches, maxHashtagLength) = (windowSize * 4, 8)

  // repeat hashtags a bit so counts are bigger than 1
  val tweets = for {
    hashtags <- Gen.listOfN(6, hashtag(maxHashtagLength))
    tweets <- BatchGen.ofNtoM(5, 10,
                addHashtag(Gen.oneOf(hashtags))(tweet(noHashtags=true)))
```

```
  } yield tweets
  val emptyTweetBatch = Batch.empty[Status]
  val gen = BatchGen.always(tweets, numBatches) ++
            BatchGen.always(emptyTweetBatch, windowSize*2)

  val alwaysEventuallyZeroCount = alwaysF[U] { case (statuses, _) =>
    val hashtags = getExpectedHashtagsForStatuses(statuses)
    laterR[U] { case (_, counts) =>
      val countsForStatuses =
        hashtags
          .map((_, ()))
          .join(counts)
          .map{case (hashtag, (_, count)) => count}
      countsForStatuses should foreachRecord { _ == 0}
    } on windowSize*3
  } during numBatches

  forAllDStream(gen)(
    TweetOps.countHashtags(batchInterval, windowSize)(_))(
    alwaysEventuallyZeroCount)
}
```

One example of the second shape of liveness properties, that use an implication in the body of an always, is the following property for `getTopHashtag`, that checks that if we superpose two generators, one for a random noise of hashtags that have a small number of occurrences (generator `tweets`), and another for a periodic peak of a random hashtag that suddenly has a big number of occurrences (generator `tweetsSpike`), then each time a peak happens then the corresponding hashtag eventually becomes the top hashtag.

```
def alwaysPeakImpliesEventuallyTop = {
  type U = (RDD[Status], RDD[String])
  val windowSize = 2
  val sidesLen = windowSize * 2
  val numBatches = sidesLen + 1 + sidesLen
  val maxHashtagLength = 8
  val peakSize = 20

  val emptyTweetBatch = Batch.empty[Status]
  val tweets =
    BatchGen.always(
      BatchGen.ofNtoM(5, 10,
                      tweetWithHashtagsOfMaxLen(maxHashtagLength)),
      numBatches)
  val popularTweetBatch = for {
    hashtag <- hashtag(maxHashtagLength)
    batch <-  BatchGen.ofN(peakSize, tweetWithHashtags(List(hashtag)))
  } yield batch
  val tweetsSpike = BatchGen.always(emptyTweetBatch, sidesLen) ++
                        BatchGen.always(popularTweetBatch, 1) ++
                        BatchGen.always(emptyTweetBatch, sidesLen)
  // repeat 6 times the superposition of random tweets
  // with a sudden spike for a random hastag
  val gen = Gen.listOfN(6, tweets + tweetsSpike).map{_.reduce(_++_)}

  val alwaysAPeakImpliesEventuallyTop = alwaysF[U] { case (statuses, _) =>
    val hashtags = getExpectedHashtagsForStatuses(statuses)
    val peakHashtags = hashtags.map{(_,1)}.reduceByKey{_+_}
          .filter{_._2 >= peakSize}.keys.cache()
    val isPeak = Solved[U] { ! peakHashtags.isEmpty }
    val eventuallyTop = laterR[U] { case (_, topHashtag) =>
      topHashtag must beEqualAsSetTo(peakHashtags)
    } on numBatches
```

```
    isPeak ==> eventuallyTop
  } during numBatches * 3

  forAllDStream(
    gen)(
    TweetOps.getTopHashtag(batchInterval, windowSize)(_))(
    alwaysAPeakImpliesEventuallyTop)
}
```

The consume operator is also useful to define other types of properties like the following, that only uses consume and next as temporal operators, but that is able to express the basic condition for counting correctly and on time. It states that for any number of repetitions n less or equal to the counting window size, and for any random word prefix, if we repeat the word prefix n times then after the $n-1$ instants we will have a count of at least (to account for hashtags randomly generated twice) n for all the hashtags in the first batch. Here we use `def next[T](times: Int)(phi: Formula[T])` that returns the result of applying next `times` times on the given formula.

```
def forallNumRepetitionsLaterCountNumRepetitions = {
  type U = (RDD[Status], RDD[(String, Int)])
  val windowSize = 5
  val (numBatches, maxHashtagLength) = (windowSize * 6, 8)

  // numRepetitions should be <= windowSize, as in the worst case each
  // hashtag is generated once per batch before being repeated using
  // Prop.forAllNoShrink because sscheck currently does not support shrinking
  Prop.forAllNoShrink(Gen.choose(1, windowSize)) { numRepetitions =>
    val tweets = BatchGen.ofNtoM(5, 10,
                                 tweetWithHashtagsOfMaxLen(maxHashtagLength))
    val gen = for {
      tweets <- BatchGen.always(tweets, numBatches)
      // using tweets as a constant generator, to repeat each generated
      // stream numRepetitions times
      delayedTweets <- PDStreamGen.always(tweets, numRepetitions)
    } yield delayedTweets

    val laterCountNumRepetitions = nextF[U] { case (statuses, _) =>
      val hashtagsInFirstBatch = getExpectedHashtagsForStatuses(statuses)
      // -2 because we have already consumed 1 batch in the outer nextF, and
      // we will consume 1 batch in the internal now
      next(max(numRepetitions-2, 0))(now { case (_, counts) =>
        val countsForHashtagsInFirstBatch =
          hashtagsInFirstBatch
            .map((_, ()))
            .join(counts)
            .map{case (hashtag, (_, count)) => count}
        countsForHashtagsInFirstBatch should foreachRecord { _ >= numRepetitions }
      })
    }
    forAllDStream(
      gen)(
      TweetOps.countHashtags(batchInterval, windowSize)(_))(
      laterCountNumRepetitions)
  }
}
```

3.3 Some additional details about the implementation

As seen in the examples from Section 2.1, there are additional factory methods in `Formula` for using the time argument in the consume operator, which are easily implemented by using the overload of `DStream.foreachRDD` that passes a `Time` object besides the current micro batch in the argument callback. Also, in the first liveness property from Section 3.2.4 we use some factory functions from `Formula` for abbreviating the combination of a consume operator with other temporal operators. The case class `BindNext` represents the consume operator, but it is mostly hidden to the

user through factories of `Formula` like `def now[T](letterToResult: T => Result): BindNext[T] = BindNext(letterToResult)`, used for applications of consume that return a timeless formula, and `def next[T](letterToFormula: T => Formula[T] = BindNext.fromAtomsConsumer(letterToFormula)`, used for applications of consume that return an arbitrary formula. The property is more intuitive when considering that the result should hold immediately after consuming the letter, i.e. *now*, and that is not incorrect when the resulting formula is timeless, because it will hold irrespectively of the input word. Besides, the implementation also stops as soon as the current formula is in a solved state, without waiting for the next letter/RDD. It is also natural to use overloads of the `next` factory both for `BindNext` and the case class `Next` for the next operator because, as seen in Section 2.1, consume is basically a capturing version of next. This way `alwaysF` is defined as `def alwaysF[T](letterToFormula: T => Formula[T]): TimeoutMissingFormula[T] = always(next(letterToFormula))`.

It is also worth mentioning some details about the way we have implemented the computation of the next form. Instead of using `Formula` to represent formulas in next form, we use a trait `NextFormula` that extends `Formula`, and a parallel type hierarchy with classes `NextNot`, `NextAnd`, etc., and even `NextNext`. This might sound at first like a bad design, but it has a number of advantages. Considering the recursive next transformation (Definition 3), we can see how those sub-formulas that cannot be evaluated in the current instant are nested under an application of the next operator. We exploit this by using a lazy argument in the constructor of the class `NextNext[T](_phi: => NextFormula[T])` that represents applications of the next operator that are in next form. The transformation is implemented as an abstract member `def nextFormula: NextFormula[T]` of `Formula`, where the recursive calls to `nextFormula` are postponed for the arguments of `NextNext`. This way those sub-formulas are only built in the moment they can be evaluated, which improves the memory behavior of the program, and allows us to mitigate the effect of the explosion in the size of the formulas due to using large timeouts. The `BindNext` class that represents an application of the consume operator $\lambda^o_x.\varphi$ also builds the next form for φ lazily, after instantiating φ when values for x and o become available.

We perform another optimization in the classes `NextOr` and `NextAnd` that represent applications of the $\vee$ and $\wedge$ operators in next form, respectively. These classes extend `NextBinaryOp` that takes a parallel collection of `NextFormula` objects—using Scala's `ParSeq` trait—that will be reduced in parallel to a result by applying a commutative and associative operator like $\vee$ or $\wedge$ [Prokopec et al., 2011]. One might think that this parallel evaluation on the Spark driver is unnecessary, because Spark is already a distributed processing engine. However, each call to a Spark action implied by assertions like those defined by `foreachRecord` or `existsRecord` blocks the calling thread, so in a sequential implementation of `NextOr` and `NextAnd` we would only be able to submit a single Spark job at a time. Our parallel implementation on the other hand is able to invoke several actions corresponding to assertions in parallel, thus taking more advantage to the Spark distributed execution, that is saturated of tasks execution requests, which improves the performance. Note this parallel sequences are frequent due to the disjunctions and conjunctions introduced by the transformation to next form.

Finally, we implement the safe word length computation from Definition 5 as a method `safeWordLength: Option[Timeout]` of `Formula`. As discussed in that definition, the safe word length can only be computed statically for some classes of formulas, that in sscheck is the class of formulas such that in all sub-formulas of the form $\lambda^o_x.\varphi$ we have that φ is a basic literal. As all bound variables are introduced by usages of the consume operator, that implies no variable applies in a timer for a temporal operator and therefore we can apply Definition 5. To do that we use a custom subclass of `scala.Function` in the factories of `Now` to mark that we can safely assume that evaluating that application of the consume operator will only take a single letter, and we return `None` in other case.

Regarding performance, on an Intel Core i7-3517U dual core 1.9 GHz and 8 GB RAM the test suite for AMP Camp's Twitter tutorial completes with success in around 22 minutes, with Spark running in local mode. That is a reasonable time for an integration test, and could be used as an automated validation step in a continuous integration pipeline [Fowler and Foemmel, 2006]. sscheck local execution could be also used for local developing to fix a broken test, using a longer batch interval configuration and smaller number of passing test cases to adapt to an scenario with less computing resources. On the other hand, if a cluster is available, sscheck could be executed using Spark distributed mode —by setting the `sparkMaster` field appropriately—, using a shorter batch interval, higher default parallelism, and a higher number of passing tests. In the future we also plan to develop a new feature to allow several test cases for the same property to be execute in parallel. This is not trivial because Spark is limited to a single Spark context per Java virtual machine (JVM)[12] but that feature, combined with a test runner

[12] `https://issues.apache.org/jira/browse/SPARK-2243`

that creates multiple JVMs,[13] would be very useful to decrease the testing time, specially if a cluster was available for testing.

A previous version of sscheck without the consume operator and the lazy optimization for next form was presented in [Riesco and Rodríguez-Hortalá, 2016b]. sscheck has also be present in leading industry conferences in the Big Data fields either presented by us [Riesco and Rodríguez-Hortalá, 2016a] or by others [Karau, 2015], and it has also been referred in books and technical blogs remarkable in the field [Holden Karau, 2015b, Karau and Warren, 2017], showing that it presents a good performance and that it stands as an alternative choice for state-of-the-art testing frameworks.

4 Related work

At first sight, the system presented in this paper can be considered an evolution of the data-flow approaches for the verification of reactive systems developed in the past decades, exemplified by systems like Lustre [Halbwachs, 1992] and Lutin [Raymond et al., 2008]. In fact, the idea underlying both stream processing systems and data-flow reactive systems is very similar: processing a potentially infinite input stream while generating an output stream. Moreover, they usually work with formulas considering both the current state and the previous ones, which are similar to the "forward" ones presented here. There are, however, some differences between these two approaches, being an important one that sscheck is executed in a parallel way using Spark.

Lustre is a programming language for reactive systems that is able to verify safety properties by generating random input streams. The random generation provided by sscheck is more refined, since it is possible to define some patterns in the stream in order to verify some behaviors that can be omitted by purely random generators. Moreover, Lustre specializes in the verification of critical systems and hence it has features for dealing with this kind of systems, but lacks other general features as complex data-structures, although new extensions are included in every new release. On the other hand, it is not possible to formally verify systems in sscheck; we focus in a lighter approach for day-to-day programs and, since it supports all Scala features, its expressive power is greater. Lutin is a specification language for reactive systems that combines constraints with temporal operators. Moreover, it is also possible to generate test cases that depend on the previous values that the system has generated. First, these constraints provide more expressive power than the atomic formulas presented here, and thus the properties stated in Lutin are more expressive than the ones in sscheck. Although supporting more expressive formulas would be an interesting subject of future work, in this work we have focused on providing a framework where the properties are "natural" even for engineers who are not trained in formal methods; once we have examined the success of this approach we will try to move into more complex properties. Second, our framework completely separates the input from the output, and hence it is not possible to share information between these streams. Although sharing this information is indeed very important for control systems, we consider that stream processing systems usually deal with external data and hence this relation is not so relevant for the present tool. Finally, note that an advantage of sscheck consists in using the same language for both programming and defining the properties.

In a similar note, we can consider runtime monitoring of synchronous systems like Lola [D'Angelo et al., 2005], a specification language that allows the user to define properties in both past and future LTL. Lola guarantees bounded memory for monitoring and allows the user to collect statistics at runtime. On the other hand, and indicated above, sscheck allows to implement both the programs and the test in the same language and provides PBT, which simplifies the testing phase, although actual programs cannot be traced.

TraceContract [Barringer and Havelund, 2011] is a Scala library that implements a logic for analyzing sequences of events (traces). That logic is a hybrid between state machines and temporal logic, that is able to express both past time and future time temporal logic formulas, and that allows a form of first order quantification over events. The logic is implemented as a shallow internal DSL, just we do for LTL_{ss} in sscheck, and it also supports step-wise evaluation of traces so it can be used for online monitoring of a running system, besides evaluating recorded execution traces. On the other hand TraceContract is not able to generate test cases, and it is not integrated with any standard testing library like Specs2.

Regarding testing tools for Spark, the most clear precedent is the unit test framework Spark Test Base [Holden Karau, 2015b], which also integrates ScalaCheck for Spark but only for Spark core. To the best of out knowledge, there is no previous library supporting property-based testing for Spark Streaming.

[13] E.g. SBT support for forking `http://www.scala-sbt.org/0.13/docs/Forking.html`

5 Conclusions and future work

In this paper we have presented sscheck, a property-based testing tool for Spark Streaming programs. sscheck allows the user to define generators and to state properties using LTL_{ss}, an extension of Linear Temporal Logic with timeouts in temporal operators and a special operator for binding the current batch and time. This logic allows a stepwise transformation that only requires/generates the current batch; using this quality the Scala implementation of sscheck takes advantage of lazy functions to efficiently implement the tool. The benchmarks presented in the paper show that the approach works well in practice. With these features in mind, we hope sscheck will be accepted by the industry; we consider the presentation at Apache Europe [Riesco and Rodríguez-Hortalá, 2016a], and citations in books written by remarkable members of the Spark community [Karau and Warren, 2017], are important steps in this direction.

There are many open lines of future work. First, adding support for arbitrary nesting of ScalaCheck `forall` and `exists` quantifiers inside LTL_{ss} formula would be an interesting extension. Moreover, we also consider developing versions for other languages with Spark API, in particular Python, or supporting other SPS, like Apache Flink [Carbone et al., 2015b] or Apache Bean [Akidau et al., 2015]. This would require novel solutions, as these systems are not based on synchronous micro-batching but they process events one at a time, and also have interesting additional features like the capability for handling different event time characteristics for supporting out of order streams, and several types of windows [Akidau et al., 2015]. Besides, we plan to explore whether the execution of several test cases in parallel minimize the test suite execution time. Finally, we intend to explore other formalisms for expressing temporal and cyclic behaviors [Wolper, 1983].

References

[Akidau et al., 2013] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044.

[Akidau et al., 2015] Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., et al. (2015). The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803.

[Alur and Henzinger, 1994] Alur, R. and Henzinger, T. A. (1994). A really temporal logic. *J. ACM*, 41(1):181–204.

[Apache Spark Team, 2016] Apache Spark Team (2016). Spark programming guide. `https://spark.apache.org/docs/latest/programming-guide.html`.

[Barringer and Havelund, 2011] Barringer, H. and Havelund, K. (2011). Tracecontract: A scala DSL for trace analysis. In Butler, M. J. and Schulte, W., editors, *Proceedings of the 17th International Symposium on Formal Methods, FM 2011*, volume 6664 of *Lecture Notes in Computer Science*, pages 57–72. Springer.

[Bauer et al., 2006] Bauer, A., Leucker, M., and Schallhart, C. (2006). Monitoring of real-time properties. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. Springer.

[Bauer et al., 2007] Bauer, A., Leucker, M., and Schallhart, C. (2007). The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification*, pages 126–138. Springer.

[Beck, 2003] Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

[Blackburn et al., 2006] Blackburn, P., van Benthem, J., and Wolter, F., editors (2006). *Handbook of Modal Logic*. Elsevier.

[Carbone et al., 2015a] Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., and Tzoumas, K. (2015a). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 38(4):11.

[Carbone et al., 2015b] Carbone, P., Fóra, G., Ewen, S., Haridi, S., and Tzoumas, K. (2015b). Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*.

[Claessen and Hughes, 2011] Claessen, K. and Hughes, J. (2011). QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64.

[D'Angelo et al., 2005] D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S., and Manna, Z. (2005). LOLA: runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, TIME 2005*, pages 166–174. IEEE Computer Society.

[Fitting and Mendelsohn, 1998] Fitting, M. and Mendelsohn, R. L. (1998). *First-Order Modal Logic*, volume 277 of *Science & Business Media*. Springer.

[Fowler, 2007] Fowler, M. (2007). Mocks aren't stubs. `https://martinfowler.com/articles/mocksArentStubs.html`.

[Fowler and Foemmel, 2006] Fowler, M. and Foemmel, M. (2006). Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, page 122.

[Gorawski et al., 2014] Gorawski, M., Gorawska, A., and Pasterak, K. (2014). A survey of data stream processing tools. In *Information Sciences and Systems 2014*, pages 295–303. Springer.

[Halbwachs, 1992] Halbwachs, N. (1992). *Synchronous programming of reactive systems*. Number 215 in Springer International Series in Engineering and Computer Science. Kluwer Academic Publishers.

[Holden Karau, 2015a] Holden Karau (2015a). Spark-testing-base. `http://spark-packages.org/package/holdenk/spark-testing-base`.

[Holden Karau, 2015b] Holden Karau (2015b). Spark-testing-base. `http://blog.cloudera.com/blog/2015/09/making-apache-spark-testing-easy-with-spark-testing-base/`.

[Kaczanowski, 2012] Kaczanowski, T. (2012). *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski.

[Karau, 2015] Karau, H. (2015). Effective testing of spark programs and jobs. In *Strata + Hadoop World 2015 NYC*. O'Reilly. `https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/42993`.

[Karau and Warren, 2017] Karau, H. and Warren, R. (2017). *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly Media, Incorporated.

[Kreps, 2014] Kreps, J. (2014). *I Heart Logs: Event Data, Stream Processing, and Data Integration*. O'Reilly Media.

[Kuhn and Allen, 2014] Kuhn, R. and Allen, J. (2014). Reactive design patterns.

[Leucker and Schallhart, 2009] Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303.

[Mackinnon et al., 2001] Mackinnon, T., Freeman, S., and Craig, P. (2001). Endo-testing: unit testing with mock objects. *Extreme programming examined*, 1:287–302.

[Marz and Warren, 2015] Marz, N. and Warren, J. (2015). *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.

[Meszaros, 2007] Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Pearson Education.

[Neumeyer et al., 2010] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE.

[Nilsson, 2014] Nilsson, R. (2014). *ScalaCheck: The Definitive Guide*. IT Pro. Artima Incorporated.

[Papadakis and Sagonas, 2011] Papadakis, M. and Sagonas, K. (2011). A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 39–50. ACM.

[Pnueli, 1986] Pnueli, A. (1986). *Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends*. Springer.

[Prokopec et al., 2011] Prokopec, A., Bagwell, P., Rompf, T., and Odersky, M. (2011). A generic parallel collection framework. In *European Conference on Parallel Processing*, pages 136–147. Springer.

[Ramasamy, 2015] Ramasamy, K. (2015). Flying faster with twitter heron. The Official Twitter Blog. `https://blog.twitter.com/2015/flying-faster-with-twitter-heron`.

[Raymond et al., 2008] Raymond, P., Roux, Y., and Jahier, E. (2008). Lutin: A language for specifying and executing reactive scenarios. *EURASIP J. Emb. Sys.*, 2008.

[Riesco and Rodríguez-Hortalá, 2017] Riesco, A. and Rodríguez-Hortalá, J. (2015–2017). sscheck: Scalacheck for spark v0.3.2. `https://github.com/juanrh/sscheck/releases/tag/0.3.2`. See ScalaDoc documentation at `https://juanrh.github.io/doc/sscheck/scala-2.10/api`, and basic setup instructions at `https://github.com/juanrh/sscheck/wiki/Quickstart`.

[Riesco and Rodríguez-Hortalá, 2016a] Riesco, A. and Rodríguez-Hortalá, J. (2016a). Property-based testing for spark streaming. In *Apache Big Data Europe 2016*. The Linux Foundation. `http://events.linuxfoundation.org/events/apache-big-data-europe/program/schedule`.

[Riesco and Rodríguez-Hortalá, 2016b] Riesco, A. and Rodríguez-Hortalá, J. (2016b). Temporal random testing for spark streaming. In Abraham, E. and Huisman, M., editors, *Proceedings of the 12th International Conference on integrated Formal Methods, iFM 2016*, volume 9681 of *Lecture Notes in Computer Science*. Springer.

[Schelter et al., 2013] Schelter, S., Ewen, S., Tzoumas, K., and Markl, V. (2013). All roads lead to Rome: optimistic recovery for distributed iterative data processing. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 1919–1928. ACM.

[Shamshiri et al., 2015] Shamshiri, S., Rojas, J. M., Fraser, G., and McMinn, P. (2015). Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1367–1374. ACM.

[Smullyan, 1995] Smullyan, R. M. (1995). *First-order logic*. Courier Corporation.

[Torreborre, 2014] Torreborre, E. (2014). Specs2 user guide. `https://etorreborre.github.io/specs2/guide/SPECS2-3.6.2/org.specs2.guide.UserGuide.html`.

[Venners, 2015] Venners, B. (2015). Re: Prop.exists and scalatest matchers. `https://groups.google.com/forum/#!msg/scalacheck/Ped7joQLhnY/gNH0SSWkKUgJ`.

[Wolper, 1983] Wolper, P. (1983). Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99.

[Yamamoto, 2010] Yamamoto, Y. (2010). Twitter4j-a java library for the twitter api.

[Zaharia et al., 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association.

[Zaharia et al., 2013] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 423–438. ACM.

A Proofs

We present in this sections the proofs for the theorems presented in the paper.

Theorem 1. *Given a formula $\varphi \in LTL_{ss}$ such that φ does not contain variables in temporal connectives, we have $nt(\varphi) = nt^e(\varphi)$.*

Proof. We prove it by induction on the structure of the formula. The base cases are the formulas for $\top$, $\bot$, terms, atomic propositions, and equalities, that are not modified and hence the property holds.

Then, it is easy to see that the property holds for the formulas defining and, or, implication, next, and consume just by applying the induction hypothesis, since both functions apply the same transformation.

Finally, we need to apply induction on the time used in temporal connectives. We present the proof for the always connective; the rest of them follow the same schema. For the base case we have:

- $nt(\Box_1\varphi) = nt(\varphi)$.
- $nt^e(\Box_1\varphi) = nt^e(\varphi) \wedge X^0 nt^e(\varphi) = nt^e(\varphi)$

This case holds by induction hypothesis in the structure of the formula. Then, assuming $nt(\Diamond_t\varphi) = nt^e(\Diamond_t\varphi)$, with $t \geq 2$, we need to prove $nt(\Diamond_{t+1}\varphi) = nt^e(\Diamond_{t+1}\varphi)$.

$$\begin{array}{lll} nt(\Box_{t+1}\varphi) & = & nt(\varphi) \wedge X\, nt(\Box_t\varphi) \\ & =^{HI} & nt(\varphi) \wedge X\, nt^e(\Box_t\varphi) \\ & =^{HI(struct)} & nt^e(\varphi) \wedge X\, nt^e(\Box_t\varphi) \\ & =^{def} & nt^e(\varphi) \wedge X(nt^e(\varphi) \wedge X\, nt^e(\varphi) \wedge \ldots \wedge X^{t-1} nt^e(\varphi)) \\ & = & nt^e(\varphi) \wedge X\, nt^e(\varphi) \wedge \ldots \wedge X^t nt^e(\varphi) \end{array}$$

□

Lemma 1 *Given an alphabet Σ and formulas $\varphi, \varphi' \in LTL_{ss}$, if $\forall u \in \Sigma^*.u, 1 \vDash \varphi \iff u, 1 \vDash \varphi'$ then $\forall u \in \Sigma^*, \forall n \in \mathbb{N}^+.u, n \vDash \varphi \iff u, n \vDash \varphi'$.*

Proof. Since $u \equiv a_1 \ldots a_m$, $m \in \mathbb{N}$, we distinguish the cases $n > m$ and $n \leq m$:

$n > m$ It is easy to see for all possible formulas that only ? can be obtained, so the property trivially holds.

$n \leq m$ Then we have $u' \equiv a_n \ldots a_m$ and, since we know that $u' \vDash \varphi \iff u' \vDash \varphi'$, the property holds.

□

Theorem 2. *Given an alphabet Σ, an interpretation $\mathcal{A}$, and formulas $\varphi, \varphi' \in LTL_{ss}$, such that $\varphi' \equiv nt(\varphi)$, we have $\forall u \in (\Sigma \times \mathbb{N})^*.u \vDash^{\mathcal{A}} \varphi \iff u \vDash^{\mathcal{A}} \varphi'$.*

Proof. We apply induction on formulas.

Base case. It is straightforward to see that the result holds for the constants $\top$ and $\bot$ and for an atomic predicate p.

Induction hypothesis. Given the formulas $\varphi_1, \varphi_2, \varphi_1', \varphi_2' \in sstl$, such that $\varphi_1' \equiv nt(\varphi_1)$ and $\varphi_2' \equiv nt(\varphi_2)$, we have $\forall u \in \Sigma^*.u \vDash \varphi_i \iff u \vDash \varphi_i'$, $i \in \{1, 2\}$.

Inductive case. We distinguish the different formulas in LTL_{ss}:

- For the formulas $\bot, \top, p, \neg\varphi_1, \varphi_1 \vee \varphi_2, \varphi_1 \wedge \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$ is straightforward to see that the result holds, since the same operators are kept and the subformulas are equivalent by hypotheses.
- For the formula $t_1 = t_2$ is straightforward to see that the result holds, since it remains unchanged.
- For the formula $\lambda_x^o.\varphi$ is also straightforward, since by hypothesis the subformula is equivalent and then the same variables are bound.
- Given the formula $X\varphi_1$, we have to prove that $\forall u \in \Sigma^*.u \vDash X\varphi_1 \iff u \vDash X\varphi_1'$. This expression can be transformed using the definition for the satisfaction for the next operator into $\forall u \in \Sigma^*.u, 2 \vDash \varphi_1 \iff u, 2 \vDash \varphi_1'$, which holds by hypothesis and Lemma 1.
- Given the formula $\Diamond_t\varphi_1$, $t \in \mathbb{N}^+$, we have to prove that $\forall u \in \Sigma^*.u \vDash \Diamond_t\varphi_1 \iff u \vDash \varphi_1' \vee X\varphi_1' \vee \ldots \vee X^{t-1}\varphi_1'$. We distinguish the possible values for $u \vDash \Diamond_t\varphi_1$:
  - $u \vDash \Diamond_t\varphi_1 : \top$. In this case the property holds because there exists i, $1 \leq i \leq t$ such that $u, i \vDash \varphi_1 : \top$. Hence, $u \vDash X^{i-1}\varphi_1'$ by hypothesis and the definition of the next operator (note that for $i = 1$ we just have $u \vDash \varphi'$).
  - $u \vDash \Diamond_t\varphi_1 : \bot$. In this case $\forall i, 1 \leq i \leq t, u, i \vDash \varphi_1 : \bot$, so we have $u \vDash X^{i-1}\varphi_1' : \bot$ for $1 \leq i \leq t$ and the transformation is also evaluated to $\bot$.

- $u \vDash \Diamond_t \varphi_1 : ?$. In this case we have u of length n, $n < t$, and $\forall i, 1 \leq i \leq n, u, i \vDash \varphi_1 : \bot$. Hence, we have $u \vDash X^{i-1}\varphi_1' : \bot$ for $1 \leq i \leq n$ and $u \vDash X^{j-1}\varphi_1' : ?$ for $n+1 \leq j \leq t$. Hence, we have $\bot \vee \ldots \vee \bot \vee ? \vee \ldots \vee ? = ?$ and the property holds.

- The analysis for $\Box_t \varphi_1$ is analogous to the one for $\Diamond_t \varphi_1$.

- Given the formula $\varphi_1 \; U_t \; \varphi_2$, $t \in \mathbb{N}^+$, we have to prove that $\forall u \in \Sigma^*. u \vDash \varphi_1 \; U_t \; \varphi_2 \iff u \vDash \varphi_2' \vee (\varphi_1' \wedge X\varphi_2') \vee \ldots \vee (\varphi_1' \wedge X\varphi_1' \wedge \ldots \wedge X^{t-2}\varphi_1' \wedge X^{t-1}\varphi_2')$. We distinguish the possible values for $u \vDash \varphi_1 \; U_t \; \varphi_2$:

  - $u \vDash \varphi_1 \; U_t \; \varphi_2 : \top$. In this case we have from the definition that $\exists i, 1 \leq i \leq t$ such that $u, i \vDash \varphi_2 : \top$ and $\forall j, 1 \leq j < i, u, j \vDash \varphi_1 : \top$. Hence, applying the induction hypothesis we have $u \vDash \varphi_1' \wedge X\varphi_1' \wedge \ldots X^{i-2}\varphi_1' \wedge X^{i-1}\varphi_2' : \top$, and hence the property holds.
  - $u \vDash \varphi_1 \; U_t \; \varphi_2 : \bot$.
    - Case a) $\forall i, 1 \leq i \leq t. u, i \vDash \varphi_2 : \bot$. In this case we have $\forall i, 1 \leq i \leq t, u, i \vDash X^{i-1}\varphi_2' : \bot$, and hence the complete formula is evaluated to $\bot$.
    - Case b) $\exists i, 1 \leq i \leq t, \forall j, 1 < j \leq i. u, j \vDash \varphi_1 : \top$, $u, j \vDash \varphi_2 : \bot$ $u, i \vDash \varphi_1 : \bot$, and $u, i \vDash \varphi_2 : \bot$. In this case we have $\forall k, 0 \leq k < i, u \vDash X^k\varphi_2' : \bot$ and $u \vDash X^{i-1}\varphi_1' : \bot$ by inductive hypothesis. Hence, all the conjunctions are evaluated to $\bot$ and the property holds.
  - $u \vDash \varphi_1 \; U_t \; \varphi_2 : ?$. In this case we have u of length n, $n < t$, $\forall i, 1 \leq i \leq n, u, i \vDash \varphi_2 : \bot$, and $u, i \vDash \varphi_1 : \top$. Hence, the first i conjunctions in the transformation are evaluated to $\bot$ by the induction hypothesis, while the rest are evaluated to $?$ by the definition of the next operator and the property holds.

- The analysis for $\varphi_1 \; R_t \; \varphi_2$ is analogous to the one for $\varphi_1 \; U_t \; \varphi_2$, taking into account that formula also holds if φ_2 always holds.

□